Data-flow based Model Analysis

Approach, Implementation and Applications

Dissertation

for the degree of
Doctor of Natural Sciences (Dr. rer. nat.)

Christian Saad

University of Augsburg

Department of Computer Science

Software Methodologies for Distributed Systems

October 2014

Data-flow based Model Analysis - Approach, Implementation and Applications

Supervisor **Prof. Dr. Bernhard Bauer**, Department of Computer Science, University of Augsburg, Germany

Advisor **Prof. Dr. Alexander Knapp**, Department of Computer Science, University of Augsburg, Germany

Defense 04 February 2015

Abstract

During the past years, the modeling paradigm has become one of the predominant trends in the field of software engineering and has been embraced by industry and research alike. An important reason for this development is that models provide an intuitive yet concise way of formalizing the concepts of an application domain along with the relationships that exist between them. As a consequence, this technique now serves as an integral part of popular and widely used software design and development processes such as the Rational Unified Process (RUP) or the Model-driven Architecture (MDA) in order to characterize static and behavioral aspects of software systems. It also drives new approaches in other contexts, e.g. in testing (Model-based Testing), for the implementation and execution of business processes on the basis of high-level descriptions (Business Process Modeling) or through the provisioning of tools aimed at domain experts (Domain-specific Languages).

However, the employment of abstraction (meta) layers to construct languages dates back a lot further than the comparatively new notion of modeling: The syntax of a programming language, usually given in the form of a context-free grammar, forms the basis for parsing language expressions into their respective structural representation. In many respects, the use of metamodels as means of defining the abstract syntax of languages therefore parallels formalisms and techniques common to the area of compiler construction.

As the application fields of modeling expand to new areas where automated processing of the contained information becomes crucial to achieve the desired outcome, the demand for methods enabling advanced analyses of the modeled content increases. Since models themselves represent a layer of abstraction w.r.t. domain-specific runtime (or "real world") semantics, their elements can be subjected to a static analysis, i.e. an assessment of their static properties which are guaranteed to hold for all instances.

Contemporary techniques for model analysis, such as the widely-used Object Constraint Language (OCL), suffer from many shortcomings with respect to their expressiveness. The conceptual similarities between the domains of modeling and compiler construction gave rise to the idea of applying the powerful and well-understood methods for static validation and optimization of formal language expressions - namely attribute grammars and data-flow analysis - to models. Hence, this thesis introduces the notion of flow-based static model analysis to enable the demand-driven, context-sensitive extraction and validation of a model's static properties and evaluates the applicability of this approach in the context of several case studies.

More specifically, the major contributions of this thesis are as follows:

- By aligning the abstraction layers and comparing the inherent structure of the respective language constructs, we investigate the conceptual similarities and differences between the domains of modeling and compiler construction w.r.t. to the implementation of an approach to static model analysis. Based on the results, we develop syntax and semantics for an analysis specification language that enables the definition of attribute-based flow analyses on metamodels. These analyses can then be instantiated and computed for arbitrary models.

- Since the commonly employed fixed-point solving algorithms for data-flow equation systems (e.g. the worklist algorithm) are not applicable in this domain, we develop and evaluate different approaches for this purpose, taking into account the characteristic properties of the modeling area to provide optimized and scalable methods for executing analyses defined using the presented technique.

- In order to determine the validity of the overall concept as well as to evaluate technical properties of the proposed algorithms we developed a reference architecture and a corresponding implementation, the Model Analysis Framework (MAF). In addition to providing IDE support for the development, testing and execution of flow-based model analyses, this framework enables the augmentation of existing third-party tools with analysis capabilities.

- Finally, we present and evaluate several application scenarios in which the devised methods have been successfully applied: The first case study introduces an analysis framework for business process analysis and includes features such as validation of structural integrity, the derivation of control-flow properties and the examination of resource definition/usage relationships. Other case studies apply the analysis technique to the fields of Enterprise Architecture Management (EAM), semantically enhanced natural language processing and the validation of AUTOSAR models.

Zusammenfassung

Im Bereich des Software-Engineerings hat das Modellierungsparadigma als Technik zur formalen Beschreibung von Anwendungsdomänen in den letzten Jahren zunehmend an Bedeutung gewonnen. Diese Entwicklung lässt sich mit der Tatsache erklären, dass Modellierungstechniken ein leicht zugängliches und dennoch mächtiges Werkzeug darstellen, das es erlaubt die Eigenschaften beliebiger Domänen einfach und präzise abzubilden. In der Folge entstanden Vorgehensmodelle zur modellbasierten Softwareentwicklung - wie etwa der Rational Unified Process (RUP) oder die Model-driven Architecture (MDA) - in denen dieser Ansatz eine zentrale Rolle spielt. So werden beispielsweise Modellierungssprachen wie die UML zur Beschreibung der strukturellen Eigenschaften und der verhaltensorientieren Aspekte des zu entwickelnden Systems eingesetzt. Da der Aufbau der Modelle dabei immer einem bekannten Muster folgt, das durch das jeweilige Metamodell vorgegeben wird, können die kodierten Daten automatisiert weiterverarbeitet werden. So kann zum Beispiel durch eine Modell-zu-Text Transformation (M2T) lauffähiger Programmcode erzeugt werden. Der Einsatz von Modellierungstechniken ist allerdings keineswegs auf die Disziplin der Softwareentwicklung begrenzt. Weitere Anwendungsfelder finden sich etwa in der Beschreibung und Generierung von Testfällen (Modellbasiertes Testen) sowie dem Design und der Implementierung von Geschäftsprozessen auf Basis abstrakter Prozessmodelle (Business Process Modeling). Weiterhin kann in jüngster Zeit ein verstärkter Trend in Richtung domänenspezifischer Sprachen beobachtet werden. Dabei werden generische Techniken wie die UML von spezialisierten Modellierungssprachen abgelöst, die auf die Benutzung durch Domänenexperten zugeschnitten sind.

Eine Konsequenz aus dem Einsatz modellbasierter Technologien ist der Bedarf an geeigneten Analysemethoden, die es ermöglichen die strukturierten Informationen auszuwerten. Einsatzbereiche für diese Methoden bestehen unter anderem in der Validierung der Modelldaten sowie der Ableitung von allgemeingültigen Aussagen in Bezug auf die Semantik der jeweiligen Zieldomäne. Statische Analyseverfahren werten hierfür die syntaktischen Struktur aus, so dass garantiert werden kann, dass die ermittelten Ergebnisse für alle Laufzeitinstanzen Gültigkeit besitzen. Aktuelle Ansätze zur Analyse von Modellen wie etwa die Object Constraint Language (OCL) sind allerdings nicht immer in der Lage diese Anforderungen zu erfüllen.

Ziel dieser Arbeit ist es, eine generische Analysemethodik bereitzustellen, die die Beschränkungen existierender Techniken umgeht. Zu diesem Zweck wird untersucht, wie Konzepte aus dem Übersetzerbau - einem wichtigen Bereich der praktischen Informatik der als gut erforscht gilt - auf Modelle angewendet werden können. Ermöglicht wird dieses Vorgehen durch konzeptuelle Ähnlichkeiten zwischen diesen beiden Bereichen. So ist die Verwendung verschiedener Abstraktionsschichten zur

Definition von Sprachen und Sprachinstanzen eine zentrale Eigenschaft des Modellierungsansatzes. In vergleichbarer Weise wird die Syntax einer Programmiersprache mittels einer kontextfreien Grammatik spezifiziert, die damit die Basis für die Verarbeitung von Sprachausdrücken (Parsen und Übersetzen von Programmcode) bildet.

In vielerlei Hinsicht ähnelt daher der Einsatz von Metamodellen als Mittel zur Definition der abstrakten Syntax einer Sprache den Formalismen und Techniken die im Bereich des Übersetzerbaus Anwendung finden. Daraus entstand die Idee, die leistungsfähigen Methoden zur statischen Validierung und Optimierung formaler Sprachausdrücke - namentlich Attributgrammatiken und Datenflußanalysen - auf Modelle zu übertragen. Diese Arbeit führt hierfür das Konzept datenflußbasierter Modellanalysen ein und evaluiert die Anwendbarkeit und Vielseitigkeit dieses Ansatzes im Rahmen mehrerer Fallstudien.

Der Inhalt lässt sich wie folgt zusammenfassen:

- In einem ersten Schritt werden Gemeinsamkeiten und Unterschiede der Quell- und Zieldomäne herausgearbeitet. Dieser Vergleich beleuchtet dabei sowohl die Beziehungen zwischen den Abstraktionshierarchien als auch die Struktur der jeweiligen Sprachkonstrukte und -instanzen. Auf Basis dieser Ergebnisse werden im Folgenden Anforderungen formuliert, die für eine erfolgreiche Übertragung der Analysekonzepte auf die Modellierungsdomäne erfüllt werden müssen. Gemäß diesen Anforderungen wird eine Spezifikationssprache entworfen, mittels derer Datenflußanalysen für beliebige Modellierungssprachen definiert und für entsprechende Modelle instanziiert werden können.

- Die Fixpunktalgorithmen (z.B. der Worklist Algorithmus) mit denen Datenflußgleichungssysteme typischerweise ausgewertet werden, können aufgrund der Adaptionen des Datenflußkonzepts nicht direkt in der Modellierungsdomäne eingesetzt werden. Aus diesem Grund müssen die klassischen Verfahren entsprechend adaptiert werden.

- Um die technischen Eigenschaften der Spezifikationen und Algorithmen praktischen evaluieren zu können wird eine Referenzarchitektur entwickelt und implementiert. Das Model Analysis Framework (MAF) stellt eine IDE für die Entwicklung, das Testen und die Ausführung von flußbasierten Modellanalysen bereit. Weiterhin ist dieses Architekturkonzept darauf ausgerichtet, das Einbetten von Analysefunktionen in existierende Drittanbieterprogramme zu unterstützen.

- Zuletzt wird das beschriebene Verfahren im Kontext verschiedener Anwendungsfälle und -szenarien eingesetzt und evaluiert: Die erste Fallstudie definiert ein Analyseframework für Geschäftsprozesse. Dieses implementiert unter anderem Funktionen zur Validierung der strukturellen Integrität und zur Auswertung der *Definition/Usage*-Beziehungen zwischen Prozessressourcen. Weitere Fallstudien sind in den Bereichen des Enterprise Architecture Managements (EAM), der semantischen Sprachverarbeitung und der Validierung von AUTOSAR-Modellen angesiedelt.

Acknowledgments

I would like to thank all who supported and encouraged me during my work on this thesis and without whom this task would not have been possible or - at the very least - much less enjoyable.

First and foremost I am deeply grateful to my supervisor, Prof. Dr. Bernhard Bauer, for giving me the opportunity to conduct my research on this topic and for being open to my ideas and propositions. His helpful and friendly guidance as well as his professional expertise provided me with the motivation which made the work on this thesis an enjoyable and interesting experience.

Furthermore, I want to thank Prof. Dr. Alexander Knapp who accepted to be advisor of my thesis. His advice and valuable feedback were vital for developing the formal aspects of the approach.

Special thanks goes to all of my former and current collegues at the Software Methodologies for distributed Systems lab at the University of Augsburg for always providing a pleasant and cheerful atmosphere to work in. I am also indebted to my collegues for their time and effort devoted to supplying me with a plethora of use cases inspired by their own research work which enabled me to thoroughly evaluate and refine my ideas.

In addition, I like to thank all bachelor and diploma students as well as the project partners who assisted me in experimenting with different approaches and with whom I had fruitful discussions about my research work.

Last but not least, I am very grateful to all my friends and family for their continued support and understanding during this time. I especially want thank my parents Gerlinde and Magdi, my brother Andreas and my grandmother Antonie for their unwavering moral support in all matters.

Contents

Part I.

Static Analysis of Formal Languages

1. Introduction

In recent years, modeling technologies have become a prominent instrument in the field of computer science with a wide range of industrial applications. One reason for the increasing popularity of Model-driven Engineering (MDE) is that this method can be used to encode features of an application domain in a simple yet concise manner. A domain's concepts and the relationships between them are stored inside a model graph. Because the structure of a model is governed by a metamodel, the codified information can be processed automatically. In this sense, modeling condenses real world scenarios into representations suitable for algorithmic interpretation.

A well-established use case can be found in Model-driven Software Development (MDSD) [SVC06] where modeling languages such as the UML are used to describe the structural and behavioral properties of software systems. Here, models can be regarded as documentation of a software's internal design while at the same they support development processes through automated code generation. As [Bez05] notes, this trend reflects a paradigm shift from the object-oriented principle of *"everything is an object"* to the model-centric view that *"everything is a model"*. Well-known examples for methodologies that heavily rely on MDE principles to enhance software engineering methods include the Model-driven Architecture (MDA) [MDA] and Model-based Testing (MBT) [AD97].

Because of its versatility, modeling has applications that reach far beyond the area of traditional software development. At their core, MDE techniques are built around a multi-tiered architecture of abstraction layers where the structural composition of any model-based artifact is always governed by a metamodel residing on its parent layer. This means that the concept of abstraction is firmly ingrained in the notion of modeling. In a broader sense, this approach can therefore also be viewed as a framework that facilitates the design of custom languages for arbitrary domains, so-called domain-specific languages (DSLs). This is reflected by the employment of the term *modeling language* that mirrors the usage of *programming language*.

In practice, the most commonly used modeling framework is the OMG's Meta Object Facility (MOF) which - along with derived languages such as the UML or the Business Process Modeling Notation (BPMN) [BPMN] - has become the de-facto standard in this area. It allows language engineers to build custom metamodels for arbitrary target domains. Arguably the most popular implementation of MOF can be found in the Eclipse Modeling Framework (EMF) and its vast ecosystem of accompanying tools that support the development of domain-specific applications. For example, the Graphical Modeling Framework (GMF) can assign visual representations to language artifacts and in turn employs *code generation* steps to automatically generate corresponding graphical language editors.

It has already been suggested that analogies can be drawn to the area of compiler construction (CC) which aims to provide the theoretical background and technical support for the realization of programming languages. On a fundamental level, both MDE and CC rely on a stack of abstraction layers along with a set of complementary techniques to standardize the process of language design in their respective domain. In fact, due to the notable similarities, the alignment of both technological spaces is a subject that has received increased attention in recent years [WK05]. This realization has lead to developments such as the Xtext language workbench [XTEX] that combine methods and practices from both areas.

There is however one important aspect to language engineering that has been hitherto neglected: While the *abstract syntax* of a programming language is typically given in the form of a context-free grammar, additional restrictions may have to be enforced to guarantee that program code can be meaningfully interpreted. As an example, statically typed languages require that any assignment of a value to a variable is preceded by (and consistent to) a declaration of the variable's type. Typically, this constraint is checked statically during the compilation process (as opposed to a runtime verification). This validation requires that each statement can be examined in its respective context since its correctness depends on the instructions that precede the statement in the program's overall control-flow. Restrictions of this type are called the *static semantics* or *well-formedness* rules of a language because they cannot be encoded in the language's syntax but can nevertheless be statically verified by other means. An important characteristic of properties computed by a static analysis is that they hold for all instances. In the presented example, the result would indicate whether a variable access is correct for all possible executions of the program. Static techniques must perform an approximation of a program's runtime behavior since the exhaustive computation of its dynamic properties is a problem that is known to be undecidable (Rice's theorem).

In the field of compiler construction, two prominent methods have emerged that are commonly employed for this purpose: Attribute grammars (AGs) and data-flow analysis (DFA) enable the validation of a program's static semantics and the derivation of optimizations based on its control-flow. In both cases, the analyses themselves are defined on the language (i.e. meta) layer and run on specific representations of language expressions, namely abstract syntax trees (ASTs) and control-flow graphs. The working principle of these approaches can be described as a form of information propagation where data sets are generated locally at the graph's nodes[1] and then forwarded along its edges. This effect is achieved by modeling the data dependencies as an equation system: A node's input parameters correspond to the output of its immediate neighbors (which in turn receive their input from their respective neighbors and so on). From a global perspective this equates to an overarching information flow as locally computed results are propagated throughout the graph. An additional advantage of the DFA method in particular is its usage of fixed-point semantics to resolve cyclic equation systems caused by backward dependencies in the control-flow. If a node has multiple predecessors, either the unification or the intersection operator is applied to combine the input sets. For cyclic and alternative

[1]In control-flow graphs, the nodes usually represent program instructions while in syntax trees, they reflect the composition of statements according to the language's grammar.

paths, it is thereby possible to distinguish between routes that *may* be traversed and routes that *must* be traversed during the program's execution.

While those two methods are firmly established in compiler construction, there still exists a lack of comparable mechanisms that would enable a validation of modeling languages. For a clarification of this assessment, we consider the example of modeling languages that encode control-flow structures, such as UML Activity Diagrams. For this type of model it is commonly required that each node can be reached from the start node. Corresponding to the variable analysis described above, the *reachability* status of a node depends on its context and can be verified statically. Nevertheless, the technological space of MDE currently does not provide a unified method for specifying this restriction in an easy, non-obtrusive and consistent way.

Over time, several attempts have been made to use existing formal approaches for models analysis, e.g. by translating (meta) models into logic-based representations [MM06; SAB10]. A drawback of this approach is that, because of the involvement of two different technological spaces, this introduces a gap that can be difficult to manage on a technical level but may also lead to problems on a conceptual level as model-specific semantics have to be mapped to the logic-based systems in which the analyses are defined and executed.

A purely model-oriented solution to this problem exists in the form of OMG's Object Constraint Language (OCL). Somewhat similar to how attribute grammars extend context-free grammars, OCL expressions are annotated at metamodel elements and can then be evaluated for derived models. However, limitations of the expressiveness which complicate the implementation of some use cases, have been the subject of discussion [MV99; Baa03]. More specifically, OCL relies on static navigation statements to address model objects which makes it difficult or impossible to implement certain types of analyses, including the computation of the reachability property mentioned previously. Intuitively, this problem could be phrased as follows: *The start node is reachable. Other nodes are reachable if at least one of their direct predecessors is reachable.* Since the reachability of each node therefore depends on the status of its immediate predecessors, any suitable solving strategy would invariably need to be able to handle recursive (and potentially cyclic) dependencies.

It is easy to see that the problem statement can expressed in the form of flow equations and thus be implemented as a data-flow analysis. The recursive nature of the problem translates into the propagation of local results to succeeding nodes. By making use of fixed-point semantics, it is furthermore possible to refine analyses through static approximations. For example, to compute the dominators for Actions in an Activity Diagram, each Action simply has to add its immediate predecessors to the intersection of the predecessors' result sets. This analysis yields the "guaranteed predecessors", i.e. the nodes that will always be traversed on the route to a specific target Action independent of the chosen execution path.

Scenarios which require a context-sensitive evaluation of model objects are quite common, with applications ranging from model validation to the computation of metrics to the approximation of dynamic properties. While it would be possible to implement a custom solution for each of these cases, it is obvious that a more generalized approach would be beneficial. This observation gave rise to the idea of transferring the DFA method to the modeling domain. Because of the close conceptual ties between compiler construction and MDE, it can be expected that a careful adaption of this well-established technique would provide a powerful method for model analysis while at the same time avoiding the inherent difficulties that often result from the combination of disparate technological spaces.

In this thesis, we address this challenge by presenting a detailed description of an approach that constitutes a generic, declarative method for the specification and computation of static properties that can be derived from the structural layout of a model. Its intended target audience are language engineers responsible for developing (model-based) domain-specific languages and tooling as opposed to users of the implemented languages (who may also be developers in their respective domain). Making use of the provided facilities, language engineers can augment their products with advanced validation and information extraction capabilities through a toolset similar to the one available to compiler designers.

In the notion of attribute grammars, we define the concept of data-flow attributes which can be attached to elements in MOF-based metamodels. For derived models, these attributes can then be instantiated and evaluated using a fixed-point computation. Motivated by the goal of preventing gaps between technological spaces, this methodology has been designed in a way that ensures a close integration with common standards and technologies in the modeling domain. For this purpose, the analysis specification language itself is based on a metamodel, thereby simplifying conceptual and technical integration. The Model Analysis Framework (MAF) comprises a tooling environment that serves as a proof-of-concept demonstrator and fosters industrial usage by enabling the integration of analysis capabilities into third-party applications. This toolset was also used to evaluate the applicability and versatility of the approach in the context of several case studies from different application domains.

The remainder of this chapter details the research questions which motivated this research, presents the objectives and highlights the contributions. It is structured as follows: In Section 1.1, we study the challenges that arise when applying static analysis to modeling languages. The objectives derived from these challenges along with the concrete approach to achieve the stated goals and the resulting contributions are described in Section 1.2. The author's publications which contributed to this thesis are listed in Section 1.3. Finally, Section 1.4 outlines the overall structure and the contents of each chapter.

1.1. Problems and Challenges

In this section we examine the current situation and the practices in the area of model-driven engineering w.r.t. the topic at hand, the specification and evaluation of the static semantics of modeling languages. Since the notion of modeling permeates a wide range of different application scenarios and use cases, we have clustered the findings into several topics, each of which focuses on one particular aspect relating to the application of analysis in the modeling domain. Starting with an assessment of the current state, we identify and describe the problems that exist in the respective context in an effort to motivate the need for a unified method that is able to address these issues. From these problems, we then derive specific challenges. These will be taken into consideration in the next section to formulate concrete objectives and outline an overall approach that implements a viable solution.

Early Detection of Errors in Model-driven Engineering

Currently, the most widely-used application of the notion of MDE can be found in the area of model-driven software development where it is employed to support the execution of traditional software engineering processes. This is usually achieved through the usage of languages such as the UML which enable the description of the internal design and behavior of software systems in a formalized, well-defined fashion. On the other hand, MDE technology is also used to implement customized solutions based on domain-specific languages that provide similar features for arbitrary application domains. In both cases, it is essential that models strictly adhere to a predefined set of restrictions to ensure that they can be automatically interpreted, e.g. by code generators or by workflow engines that are able to execute business process models.

Problems

- Whether models are used to describe the inner workings of software systems or to encode information in other areas, it must be assumed that the process of modeling itself - just like any other specification process - is inherently error-prone and thus requires multiple iterations with subsequent verification steps to achieve the desired outcome. Since modeling is usually carried out in the early phases of a development process, it is vital that developers are given immediate feedback on the correctness of their models. Otherwise, early design errors might lead to larger problems later on that are costly and time-consuming to rectify.

- An often neglected aspect of the modeling process is the quality assurance of the developed artifacts. For program code, it is common practice to define and (automatically) enforce a set of coding guidelines to improve software quality and reduce the probability of errors. With the increasing importance of the role of models in many development processes, the need for a suitable technique for this purpose becomes evident. Unfortunately, modeling guidelines often exist only as an informal, textual description that cannot be used for an automated verification.

Challenge 1: Test the Models - Not the System

To ensure that problems can be identified in the early stages of a development process and to provide feedback to the developers about the quality of the design artifacts, analysis has to be performed on the model level. In other words, it should be possible to subject models to an evaluation of their static properties comparable to the validation of program code by a compiler. In a sense, this represents a "test" of the models themselves rather than of the generated code during runtime.

Analysis of a Model's Dynamic Properties

In the previous point, it has been mentioned that modeling is usually part of the design phase. Instead of working on an instance of the running system, analysis techniques therefore have to rely solely on the defined models. To generate valuable feedback, any viable method would therefore have to be able to extract information that is implicitly encoded in the design artifacts. This is complicated by the fact that the meaning of the contained elements is highly domain-specific. For example, the information encoded inside an UML Class Diagram has to be interpreted in a different way than the elements found in an Activity Diagram.

Problems

- Static analyses are defined on the language level and executed for derived expressions such as models or program code. Often, these expressions are themselves abstract representations of a - possibly infinite - set of instances: Each execution of a program can be unique in itself although it is based on the same program code. The same is true for models. A business process model, for example, describes a whole class of possible process instances. An analysis of language expressions therefore must be able to factor in characteristics that influence runtime behavior.

- Metamodels constitute the abstract syntax of modeling languages in the same way context-free grammars define the syntactical structure of programming languages. Both metamodels and CFGs are intuitive formalisms that enable an efficient algorithmic processing of derived language expressions. The downside of this approach is that both static and dynamic semantics, i.e. well-formedness rules and behavioral aspects, must be specified separately. Current solutions to this problem suffer from several drawbacks such as relying on complex theoretical abstractions, introducing technological gaps or being too limited in their expressiveness.

Challenge 2: Approximate Dynamic Behavior

During the design phase, the only available input for analysis consists of the design artifacts themselves. Results must therefore be extractable based on an interpretation of a model's structural composition. Because the semantics of model expressions depends on the respective application domain, this process has to take into account domain-specific properties of the language's elements. For this purpose, a simple, intuitive mechanism is required for the definition of a metamodel's semantics. More specifically, this pertains to the class of static properties that hold for all instances of a model and therefore enable an approximation of its "runtime behavior".

Applicability to Different Scenarios

As described earlier, the notion of modeling can be applied in a wide variety of different scenarios. Common use cases include code generation, the encoding of the structural composition of objects in a target domain, the definition of executable control-flows, data storage and others. It has already been stated in Challenge ① that the interpretation of model elements therefore depends on the specific runtime semantics. However, a closer examination of the usage of modeling techniques reveals that there are even more factors involved. For example, there is a distinction between the roles of language engineers on one side and language users on the other side. Both parties may put an emphasis on different aspects when it comes to the application of analysis techniques. In addition, it is also possible to distinguish between different goals that users might pursue such as the validation of models, an assessment of their quality or information extraction.

Problems

- Modeling is a highly versatile technology that can be used to achieve different goals. It is therefore not advisable to hard-code functionality for concrete domains such as software development or for specific modeling languages like UML. Nevertheless, some usage scenarios may be applicable in multiple settings. As a consequence, it should be possible to adapt existing analyses to work in similar domains with a comparatively minor effort.

- In the application of analysis functions, users might pursue different goals. Any technique for this task would have to be flexible enough to handle scenarios such as validation of structural integrity, an assessment of a model's semantic properties or an estimation of the model's quality.

- Many applications of MDE implicitly assume the involvement of two parties: The language developers and the language users. One therefore has to distinguish between two different viewpoints and their respective expectations when it comes to the application of analysis techniques. On the one hand, a language developer is concerned with the design of the language itself and the provisioning of supporting tooling environments while users have an interest in being able to efficiently employ the provided methods and tools.

Challenge 3: Support Domain-independence and Reusability

To support the evaluation of dynamic properties across domains and with different goals, any viable approach has to be generically applicable. This means that it must not make assumptions that presume the existence of features found only in specific target domains. In addition, the method must also be generic in the sense that it supports a range of application scenarios such as the validation of structural integrity or the extraction of dynamic properties. In this context, once defined analyses should also be adaptable to different domains that share the same basic principles. Finally, the approach has to distinguish between the roles of developers and users by providing language engineers with facilities that enable a simple specification of reusable analyses that are able to yield the information required by language users.

Consistent Integration with MDE Principles

Modeling frameworks such as MOF or KM3 [JBT06] define multi-layered architectures for the development and usage of model-based domain-specific languages. These standards are accompanied by a large set of complementary, interwoven techniques including, amongst others, model transformations, editor builders and database persistence layers. As a whole, this toolset forms the basis for the versatile application of MDE principles. Consequently, it is desirable that any new method that seeks to extend the capabilities of the modeling ecosystem should integrate with existing techniques.

Current methods that allow the definition and execution of analyses on models take various approaches to accomplish this task. While the OCL, for example, relies solely on modeling technology, other techniques translate problems into different technological spaces such as formal logic.

Problems

- The direct reuse of existing formal methods for the purpose of model analysis requires the transfer of modeling semantics to the respective target domain. In addition to the resulting technological gap, a mapping between the semantics of both domains is further complicated by concepts which are not directly translatable [Ana+10]. While this is certainly a valid approach, it would introduce problems that may hinder its successful application. In addition, users would be required to familiarize themselves with the semantics of a domain outside of their actual scope of expertise.

- Metamodels themselves are based on a meta metamodel - M3 in MOF terminology - which can be regarded as a language for language development.

One way to provide an integrated, model-based solution that allows for the annotation of analysis specifications at metamodels would be to extend the M3 layer with additional constructs [MFJ05]. However, this would likely result in an incompatibility with existing MDE methods and tools that depend on a faithful implementation of the official standards.

- Users that are familiar with the modeling domain expect that the practices in this field are respected. Constraints formalized in the Object Constraint Language, for example, are defined in the context of metamodel elements and follow the common modeling semantics of generalization and instantiation. To gain user confidence, it is therefore vital that these notions are supported in a simple and intuitive fashion.

Challenge 4: Integrate with Modeling Practices

It is often the case that a modeling technique builds upon existing methods and at the same time offers new functionality that can be reused by other techniques. The successful application of any method that seeks to extend the current state in the MDE space therefore requires a close integration with the standards and practices in this area. For this reason, rather than simply translating (meta) models to a representation suitable for flow analysis, the analysis specification process should itself follow modeling principles. To preserve compatibility with existing techniques and implementations, this has to be done in a way that is non-intrusive, i.e. does not require a modification of established standards and tools.

1.2. Objectives, Approach and Contributions

In the previous section, we identified problems and challenges relating to the implementation of an analysis technique supporting the derivation of meaningful information about a model's dynamic properties from its structural composition. Now we will take these aspects into consideration in an effort to develop concrete objectives for the realization of this method and discuss in more detail how these goals can be achieved. This will be complemented by a listing of the scientific contributions of this thesis.

Objectives

The following list comprises the major objectives of this thesis and relate to the challenges presented in Section 1.1.

Objective 1: Provide Method for Static Model Analysis

Challenge ① emphasizes the importance of performing analysis on a model level. This requires the implementation of a sound, practicable and efficient methodology that enables the extraction of dynamic properties. It is therefore necessary to provide the possibility to approximate a model's runtime semantics as noted in Challenge ②.

For this purpose, we propose the application of data-flow analysis, a well-known compiler construction technique, since this method fulfills the listed key requirements: A flow analysis is a declarative specification that can be defined for existing metamodels. Its constituents, the flow equations that compute and propagate local results, are able to generate and process context-sensitive information and can be phrased in a way which factors in the domain-specific semantics of the respective modeling language.

Through the usage of fixed-point evaluation semantics, this technique derives properties that are true for all interpretations of a model, thereby approximating dynamic behavior. Depending on the employed confluence operator, analyses can distinguish between minimal and maximal cases, i.e. facts that are guaranteed to hold for all instances or may hold for some instances respectively.

The objective therefore is to start with the DFA method from the field of compiler construction and adapt it for use in the modeling domain.

Objective 2: Ensure Generic Applicability

Challenge ③ stresses the importance of the applicability of the approach with respect to different aspects: Developers and users might choose to employ static analysis in arbitrary target domains for different purposes. The implementation of the flow analysis technique must therefore not rely on features of specific types of models. For example, it would not be safe to automatically route information along the control-flow edges of Activity Diagrams as both the presence of these edges and their concrete semantics depend on the respective modeling language. Instead, a domain-independent way of specifying the information flow in models is needed.

On a technical level, it is also important to consider the challenges that arise when attempting to implement a method that consistently integrates with the modeling domain. Challenge ④ emphasizes that this is an essential goal for any viable modeling technique. Consequently, it must be guaranteed that the methodology not only extends but also retains full compatibility with existing standards. For this reason, the analysis specification process itself has to be based on a modeling language that is both defined on the basis of the Meta Object Facility and can extend MOF-derived metamodels in a non-intrusive fashion. Simultaneously, DFA evaluation semantics must be adapted to support a flexible definition of information propagation as stated above.

To remain independent of specific implementation technologies, it should also be possible to specify the execution semantics of data-flow equations using arbitrary languages.

An additional objective related to general applicability is the provisioning of a domain-independent standard library of analysis templates. This ensures that similar problems appearing in different domains can be addressed effectively.

Objective 3: Provide Reference Implementation

Any practical application and evaluation of the conceived approach requires the development of a proof-of-concept implementation. Since the technology is intended to be employed by third parties, it is desirable that this component fulfills the demands of an industry-grade application. As a prerequisite for this implementation, it is therefore necessary to carefully devise a set of design goals and draft a suitable architecture. Furthermore, it must be ensured that support for widely used frameworks and modeling technologies is available.

In this context, it is also necessary to consider the different roles of language engineers and languages users and their respective preferences as described in Challenge ③. For developers, an IDE must be provided that implements the analysis DSL. The execution of the defined analyses is handled by a separate DFA evaluator component that can be integrated into modeling applications, thereby making analysis capabilities available to users.

Objective 4: Evaluate the Approach

To provide proof that the proposed method indeed fulfills the stated requirements, it must be subjected to an in-depth evaluation according to different criteria. In order to assess the points listed in Challenge ①, it must be demonstrated that model analysis is a viable method for extracting useful information from model artifacts in early development phases. Challenge ③ demands the applicability for different domains and purposes. To evaluate this requirement, it is necessary to study how this approach can be employed to solve problems in multiple usage scenarios. Consequently, we have to identify case studies which reflect the diverse nature of the use cases to enable the examination of the proposed key features.

Approach and Contributions

To implement a solution that is able to address the points listed in the stated objectives, multiple steps have to be carried out. We will now outline these “work packages” (which also represent the general structure of this thesis) and summarize the contributions that have been developed for the presented challenges and problems.

1. Comparison of Compiler Construction and Model-driven Engineering

As [Bra09] notes, *“[t]here is a huge amount of compiler construction related research from which the model-driven software engineering can learn”*. In the context of this thesis, this statement applies to the transfer of the concept of data-flow analysis to the modeling space.

To accomplish this task, it is first necessary to provide a sound understanding of the conceptual relationships between the techniques involved in this process. It is known that the technological spaces of compiler construction and modeling share several characteristics. A close examination of both fields will therefore reveal their conceptual similarities as well as subtle differences. This comparison is required to identify which parts of the original method can be directly transferred to the target domain and also to anticipate potential complications and to help devise proper ways to address these problems. The concrete requirements for an effective implementation of a flow-based model analysis concept can then be derived from these considerations.

While the comparison between both technological spaces and the drawn conclusions provide the underpinnings for the realization of the proposed approach, this can also be understood as an independent study of the relationships between both language frameworks and the implications for techniques that focus on non-invasive, declarative annotations of modeling languages.

Contributions

- We examine the relationships between the language frameworks of compiler construction and modeling. This includes a study of their respective applications areas, the methodologies of language development and usage and an alignment of the abstraction layers of the multi-tier architectures that are the corner stone of both domains. We also take a closer look at the role syntax and (static) semantics play in the context of domain-specific languages realized with either technique.

- From these observations, we derive and motivate a set of design goals and intended properties relating to the implementation of the proposed flow-based analysis technique. Subsequently, we concretize the resulting challenges in an effort to examine how the notion of flow-sensitive analysis can be adapted to the modeling domain in a pragmatic way while preserving its original semantics.

2. Adapt DFA Semantics to the Modeling Domain

In order to provide a solid foundation for this method, it is necessary to give a sound mathematical description of the proposed concepts. There are multiple aspects that must be considered in this process. First, the concepts required for analysis specification themselves have to be formalized based on a suitable representation of the target domain. In a second step, these definitions can then be used to describe semantics with respect to the evaluation of analyses. Finally, it must be shown that this definition is in line with traditional DFA, i.e. that it actually represents a valid transfer of this technique to the target domain.

Contributions

- We present a mathematical description of the core concepts of flow analyses. For this purpose, we define the syntax and instantiation semantics of analysis specifications in the notion of attribute grammars. With this approach, we are able to implement a declarative, non-intrusive technique that enables the extension of existing modeling languages with declarations of their static semantics. To ensure a proper integration with the target domain, we base this specification on a formalized representation of modeling concepts.

- The comparison of the technological spaces of MDE and compiler construction leads to the conclusion that traditional algorithms for solving DFA equation systems cannot be directly applied to the modeling domain. To address this challenge, we propose a set of properties that must be fulfilled by a viable adaption of this method. The contribution made in the context of this topic therefore consists of an approach that implements the required features. It represents a demand-driven, partially parallelizable algorithm for DFA fixed-point computation that supports the dynamic discovery of data-flow dependencies.

3. Develop DSL for Flow-based Model Analysis

The previous step already ensures a sound theoretical understanding of how to define and conduct flow analyses on models. However, for a practical application of this approach, this information must be complemented by the artifacts required to actually define and implement analyses for (meta) models. This means that a suitable representation of the formal concepts has to be developed.

Keeping in line with the stated objectives, the analysis technique should not only extend the current state of MDE but it should itself be firmly rooted in MDE methodology. To this end, we have to provide a custom model-based domain-specific language that realizes the proposed concepts, thereby supporting the specification of analyses for existing modeling languages.

Contributions

- We define a metamodel that implements a domain-specific language for the attribute-based specification of data-flow analyses. This approach ensures that concepts of existing modeling languages can be annotated with analysis constructs in a non-intrusive fashion. As a consequence, neither the MOF framework itself nor the target language have to be modified in any way. Because of the declarative nature of the annotations, analyses can be partitioned into libraries thereby streamlining the specification process. Also, this DSL only presumes the existence of very few basic modeling concepts - namely classes, associations and generalizations - and can therefore be considered to be a generic solution for the modeling space.

- For practical usage, the abstract syntax defined by the metamodel has to be complemented by a concrete representation. For this purpose, we present a textual DSL. The structure of this language is given in the form of a Xtext grammar since this variant of context-free grammars directly associates grammatical symbols with metamodel concepts. From this definition, the Xtext language workbench is able to automatically generate matching parsers and serializers.

4. Develop Toolset for Model Analysis

A programmatic realization of the proposed approach is a necessary prerequisite for a proper examination of its properties in practical usage scenarios.

The first step in this direction consists of an implementation of the central artifacts and concepts. More specifically, this comprises the specification language, the instantiation process and the algorithms supporting fixed-point computation.

For a practical application of the developed methodology, the functional core must be embedded in a framework that facilitates its usage in complex scenarios. To devise a suitable architectural design for this purpose, we focus on the intended use cases: The framework has to provide support for a variety of tasks such as validation and information extraction and has to anticipate the respective demands of the roles of language engineers and users. Additionally, the technologies that constitute the MDE ecosystem have to be assessed to ensure a consistent integration of the developed tools.

Contributions

- We provide a list of design goals that motivate and describe the desired features of the analysis framework. These goals detail the requirements that have to be met to ensure that the analysis tool represents a viable solution for all intended purposes. We then present a concept that incorporates these considerations. It consists of an overall architectural design and an description of its functional components and their respective tasks and interactions with other parts of the framework.

- We present our reference implementation - the Model Analysis Framework (MAF). In this context, we demonstrate how the abstract concepts of the proposed architecture can be realized to develop an integrated flow analysis toolset. This includes an IDE that provides language engineers with the ability to specify analyses and simplifies the integration of analysis capabilities into third-party applications.

5. Demonstrate Practical Applicability

At this point, the theoretical framework has been fully developed and a practical implementation exists in the form of the Model Analysis Framework. However, the approach still must be subjected to a practical evaluation to demonstrate that it fulfills the initially stated objectives.

For this purpose, multiple case studies have been carried out, each in the context of a different application domain. Each case study is itself composed of a variety of smaller use cases that address specific problems in the respective target domain and serve as building blocks for the realization of more complex goals. The intention here is to provide proof of the versatility of the proposed method with respect to its applicability to a wide range of modeling languages as well as its flexibility in handling versatile usage scenarios.

Contributions

- It has been stated that some problems that can be solved through flow analyses - such as determining the reachability of flow-graph nodes - can be found in different domains. To simplify the adaption of these recurring use cases, we provide a standard library comprised of analysis templates that can be easily adapted to work in target domains that share a set of basic properties.
- We present four larger case studies from different domains, each consisting of multiple use cases. Some of the use cases are similar to common compiler construction analyses while others represent innovative methods for deriving useful information from model-based languages such as AUTOSAR. These results constitute starting points for further research in the respective areas and are therefore also of relevance outside of the specific scope of this thesis.

1.3. Publications

Parts of this thesis have been previously published in the following peer-reviewed publications:

1. Christian Saad, Florian Lautenbacher, and Bernhard Bauer. “An Attribute-based Approach to the Analysis of Model Characteristics”. In: *Proceedings of the 1st International Workshop on Future Trends of Model-Driven Development in the context of the 11th International Conference on Enterprise Information Systems (ICEIS)*. vol. 9. FTMDD’09. 2009

   This position paper introduces and motivates the basic notion of applying techniques from the field of compiler construction to the modeling domain.

2. Christian Saad and Bernhard Bauer. “Applying Data-flow Analysis to Models - A Novel Approach for Model Analysis”. In: *Proceedings of the Spring Simulation Multiconference*. SpringSim’10. ACM, 2010, p. 241, best poster presentation award

   This contribution consists of a short paper with an accompanying poster presentation which further develops the initial proposals and describes basic analysis scenarios, some of which have been included in Chapter 9.

3. Christian Saad and Bernhard Bauer. “Data-flow Based Model Analysis”. In: *Proceedings of the 2nd NASA Formal Methods Symposium*. Vol. NASA/CP-2010-216215. NFM’10. NASA. 2010, pp. 227–231

   The short paper summarizes and clarifies the ideas from the previous publications and discusses more complex usage scenarios which are reflected in the incremental analyses in Section 10.1.

4. Christian SAAD and Bernhard BAUER. "Analyzing Dynamic Models using a Data-flow based Approach". In: *Proceedings of the 1st Doctoral Symposium in the context of the 3rd International Conference on Software Language Engineering*. SLE'10. 2010, p. 37

   A short contribution to a PhD symposium which summarizes the motivations and the research questions which are addressed in this thesis.

5. Christian SAAD and Bernhard BAUER. "The Model Analysis Framework - An IDE for Static Model Analysis". In: *Proceedings of the Industry Track of Software Language Engineering in the context of the 4th International Conference on Software Language Engineering (SLE)*. ITSLE'11. 2011

   This tooling paper discusses the Model Analysis Framework which is described in greater detail in Chapter 8 while the general considerations behind the architectural design are explained in Chapter 7.

6. Christian SAAD and Bernhard BAUER. "Data-flow based Model Analysis and its Applications". In: *Proceedings of the 16th International Conference on Model Driven Engineering Languages and Systems*. MoDELS'13. Springer-Verlag, 2013, pp. 707–723

   In this publication, we gave a more detailed overview of the developed methodology and its artifacts, providing the starting point for the descriptions in Chapter 6. This includes the abstract and concrete syntax of the analysis specification language as well as the dependency chain algorithm. Additionally, the paper illustrates the definition/usage analysis scenario (cf. Section 10.1.3).

7. Melanie LANGERMEIER, Christian SAAD, and Bernhard BAUER. "A unified Framework for Enterprise Architecture Analysis". In: *Proceedings of the Enterprise Model Analysis Workshop in the context of the 18th Enterprise Computing Conference*. EDOC'14. 2014

   This paper develops a unified framework for the analysis of models in the EAM domain. The generic representational format for (meta) model data as well as the metrics use case of the case study presented in Section 10.2 are derived from this paper.

8. Melanie LANGERMEIER, Christian SAAD, and Bernhard BAUER. "Context-sensitive Impact Analysis for Enterprise Architecture Management". In: *Proceedings of the 4th International Symposium on Business Modeling and Software Design*. BMSD'14. 2014

   This paper contributes the second use case for the analysis of EAM models, namely the computation of change impacts.

Publications 1-6 consist of research work that has been carried out by the author of this thesis while the contributions to publications 7 and 8 are limited to the ananlysis-specific aspects of the respective approaches.

The Model Analysis Framework constitutes an implementation of the proposed methodology. It is available from:

http://code.google.com/a/eclipselabs.org/p/model-analysis-framework/

1.4. Outline

The general structure of this thesis is illustrated in Figure 1.1. To ensure a comprehensive understanding of the described methodology, it is suggested that the chapters are read in the proposed order as indicated by the arrows.

To enable a quick orientation, the thesis has been divided into four major sections. The first part motivates the proposed method, presents theoretical foundations and gives an overview of related work. Readers that are already familiar with the fields of compiler construction and/or MDE may skip the respective sections of Chapter 2. The data-flow oriented approach to model analysis is described in Part II which should therefore be read before proceeding to the remaining chapters. Part III details the architecture and the usage of the reference implementation - the Model Analysis Framework - while Part IV presents multiple case studies that demonstrate the application of the analysis method. These parts can be read in any order.

The following summary gives a short overview of the contents of each chapter:

Part I - Static Analysis of Formal Languages

Chapter 1 - Introduction
This chapter presents the contextual framework for the subject matter of this thesis by stating the problem description and motivating the method that has been chosen to approach these issues. It further details the challenges that have to be addressed and derives the concrete objectives. Finally, it highlights the scientific contributions and lists the author's publications.

Chapter 2 - Basics
The contributions of this thesis are built on different techniques. More specifically, this comprises the field of compiler construction and its canonical methods for static analysis, attribute grammars and data-flow analysis. The second area of relevance is the technological space of modeling representing the target domain to which the aforementioned methods are applied. This chapter describes the technical background of these fields and details specific standards and practices which are relevant to the development of the analysis approach.

Chapter 3 - Related Work
Since the study of formal languages is a very active field in computer science, different methods and solutions have been proposed to address the problem of semantic analysis. Here, we describe notable examples of these efforts along

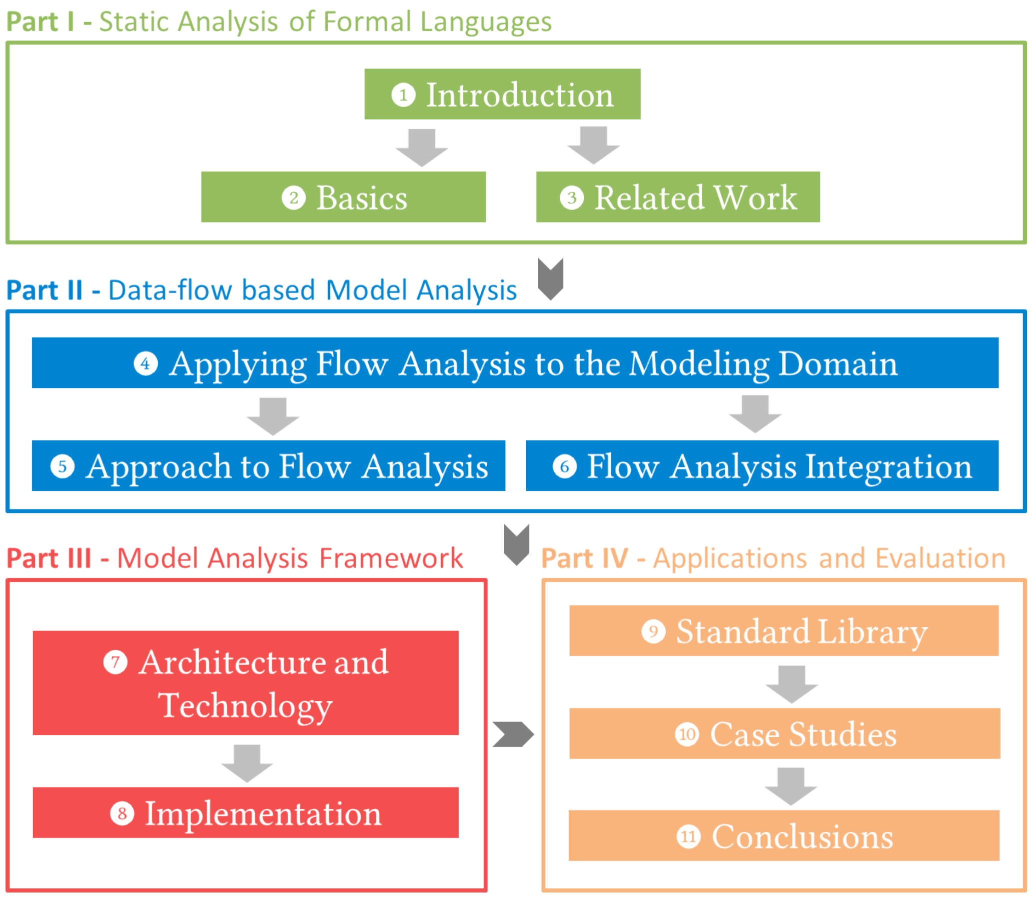

Part I - Static Analysis of Formal Languages
1 Introduction
2 Basics
3 Related Work
Part II - Data-flow based Model Analysis
4 Applying Flow Analysis to the Modeling Domain
5 Approach to Flow Analysis
6 Flow Analysis Integration
Part III - Model Analysis Framework
7 Architecture and Technology
8 Implementation
Part IV - Applications and Evaluation
9 Standard Library
10 Case Studies
11 Conclusions

Figure 1.1.: The structure of this thesis.

with their respective advantages and disadvantages in the context of model analysis and assess their relationship with the topic of this thesis.

Part II - Data-flow based Model Analysis

Chapter 4 - Adapting Flow Analysis to the Modeling Domain

The first section of Part II concretizes the notion of a technological space as a field of related standards and practices to motivate the transfer of analysis techniques from the area of compiler construction to the modeling domain. Based on this description, we develop an alignment between these two spaces to deepen the understanding of the basic principles behind these language frameworks. From this examination we derive a list of design goals for the transfer of the flow analysis concept along with a set of challenges that lie on this path. This is followed by an outline of the concrete steps that have to be taken and the required artifacts that must be developed in order to realize the proposed method.

Chapter 5 - Formal Semantics of Flow-based Model Analyses

In this section, a theoretical framework for a flow-based model analysis is developed. It consists of mathematical descriptions of the involved methods and algorithms.

Chapter 6 - Model-based Integration of Data-flow Analysis
The theoretical concepts that have been formulated in the last section are applied to the MDE domain using modeling technologies. This involves the provisioning of a suitable modeling language for the development of flow-analyses - comprised of a metamodel and a textual syntax - and its integration with the MOF standard. In addition, methods for building and computing the equation systems for the fixed-point calculation are presented.

Part III - The Model Analysis Framework

Chapter 7 - Architecture and Technology
The practical implementation of the DFA approach requires the combination of many model-based technologies. These are introduced in the first part of this chapter. Subsequently, different usage scenarios for the analysis tooling are motivated and described. More specifically, this includes the employment as a research-oriented framework for evaluating the proposed approach as well as an IDE for analysis development and its usage as a library that can be integrated into existing applications. From these applications, requirements are derived in the form of design goals and a suitable architectural design is proposed.

Chapter 8 - Implementing Flow-based Model Analysis
Based on the observations from Chapter 7, this section describes how different aspects of MAF's architecture are implemented. This comprises the central DFA evaluator component as well as an IDE intended to be used by language engineers for analysis specification. It is also discussed how analyses can be configured and integrated into third-party applications.

Part IV - Applications and Evaluation

Chapter 9 - Application Scenarios and Analysis Templates
It can be observed that - independent of the concrete application domain - many problems that can be solved using data-flow analyses bear a close resemblance to each other. This section establishes a standard library for DFA problems by providing reusable reference implementations that address recurring challenges, for example the task of analyzing control-flow structures.

Chapter 10 - Case Studies and Applications
The applicability of the proposed method is evaluated in the context of four case studies from different application domains. Each case study consists of several smaller use cases that address different usage scenarios ranging from validation to the extraction of implicitly contained static information. Through the combination of these use cases, it is demonstrated how solutions to more complex problems can be built incrementally. The presented

case studies contain an evaluation of the employed analyses and a description of their practical implementation demonstrating how this technology can be integrated into different target environments.

Chapter 11 - Conclusions and Outlook
The last section summarizes the contributions of this thesis and discusses the implications for the field of model-driven engineering. It concludes with an outlook on future work by providing starting points for the further extension and improvement of the analysis method presented in this thesis.

2. Basics

The approach that is detailed in this thesis relies on techniques from two distinct fields, the areas of compiler construction and modeling. Both domains provide tools for the definition of languages as well as methods and algorithms for the processing of derived language expressions. In this chapter, we will showcase the guiding principles and the formal underpinnings of both fields with respect to the requirements of the specification of a flow-based analysis approach. In the context of this thesis, these descriptions will serve as the basis for the examination of the properties shared by compiler construction and modeling (Section 4.1), from which we will then derive the basic design (Section 4.2) as well as the technical specification (Chapter 6) of our approach.

Section 2.1 provides an overview of the field of compiler construction, including a selection of contemporary practices and standards commonly employed for language definition and the representation of language expressions. Furthermore, this section investigates the subject of program analysis.

In the area of compiler construction, there exist two techniques which are commonly used to validate and analyze language expressions. These methods are known as attribute grammars and data-flow analysis. Attribute grammars, which are described in Section 2.2, support the annotation of language definitions with declarative analysis specifications which can then be instantiated and evaluated for arbitrary language expressions. Using data-flow analysis, it is possible to approximate the runtime behavior of programs through a fixed-point computation of cyclic equation systems. Data-flow equation systems, which are derived from the program's control-flow graph, implement the propagation of information from each program statement to its respective successors, enabling the examination of each instruction in its overall context. This concept is explained in Section 2.3.

In the field of modeling, language expressions are referred to as models, with metamodels replacing formal grammars as primary tool for specifying the syntax of a language. The principles, definitions and methods of the modeling domain are described in Section 2.4 with an emphasis on the de-facto standard in industry and research, the Meta Object Facility (MOF).

2.1. Program Translation and Analysis

This section outlines the basics of compiler design and the working principles of compilers. In Sections 2.1.1 and 2.1.2, we shortly introduce the basic notions of the compiler construction domain and formal languages respectively. Subsequently, in Section 2.1.3, we explore the properties of syntax trees and control-flow graphs, two methods commonly used by compilers for the internal representation of programs.

These formats provide the context for the application of attribute grammars (cf. Section 2.2) and data-flow analysis (cf. Section 2.3). Finally, in Section 2.1.4, we describe a canonical classification system for program analysis techniques.

2.1.1. Compiler Construction

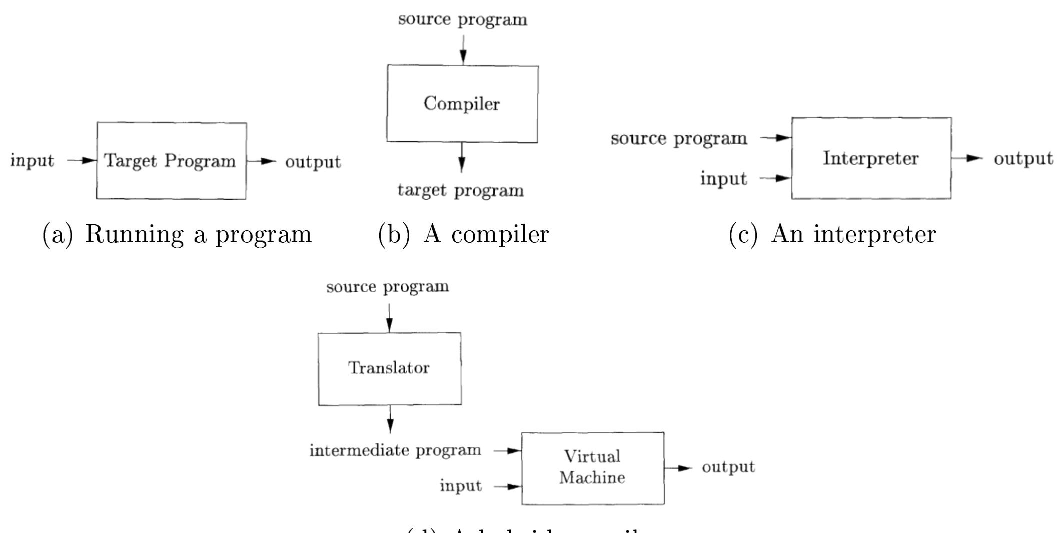
In computer science, the field of compiler construction targets the problem of translating *"a program in one language - the source language - [...] into an equivalent program in another language - the target language"* [Aho+06]. Many of its concepts are based on the theories of formal languages and automatas. In a sense, compiler construction can therefore be considered to be a practical application of theoretical computer science. In contrast to the area of computer linguistics which deals with the algorithmic processing of natural languages, the artificial languages encountered in the field of compiler construction are constrained by a set of restrictions which aim to make language expressions easily parsable by computers.

The most common use case for compilers is the translation of human-readable source code written in a programming language into machine code that can be executed by computer processors. Because programming languages and computer architectures often differ significantly in their design and the objective of realizing a efficient and robust implementation of the desired functionality is very complex, the task of writing a compiler tends to be a challenging and error-prone process. For this reason, a set of techniques and best-practices have been developed in the last decades that help in achieving this goal.

The frameworks found in the compiler construction domain include methods for the specification, representation and validation of formal languages. In addition, since the efficient usage of the available resources is an important requirement in computing, several techniques for the optimization of program code have been devised to eliminate redundancy, reduce memory consumption and make use of multicore machines for parallelized execution while preserving the execution semantics of the original program.

Because of the importance of these issues, there is much ongoing research in this area. This includes improved methods for optimization, especially with regard to the increasing availability of multicore systems [DM98; Bas+09] as well as the integration of different programming paradigms, for example the combination of object oriented and functional languages [Ode+04] and the trend towards domain-specific languages [Slo08]. However, as mentioned above, the foundations of compiler design are well-researched and the corresponding techniques are described in the canonical literature, e.g. [Aho+06; WM95; Gru+00; Mor98]. Based on these sources, we will now present some of the basic principles which will play a important role in the specification of our own approach.

Figure 2.1 outlines several methods for the execution of programs encoded in a high-level language. Independent from the respective approach - from the view point of the user - a program behaves as seen in Figure 2.1(a), i.e. it produces a specific output for a given input. Programs that are written in compiled languages such as C have to be translated into instructions for a specific target environment as shown



(a) Running a program (b) A compiler (c) An interpreter

(d) A hybrid compiler

Figure 2.1.: Compiler classification [Aho+06].

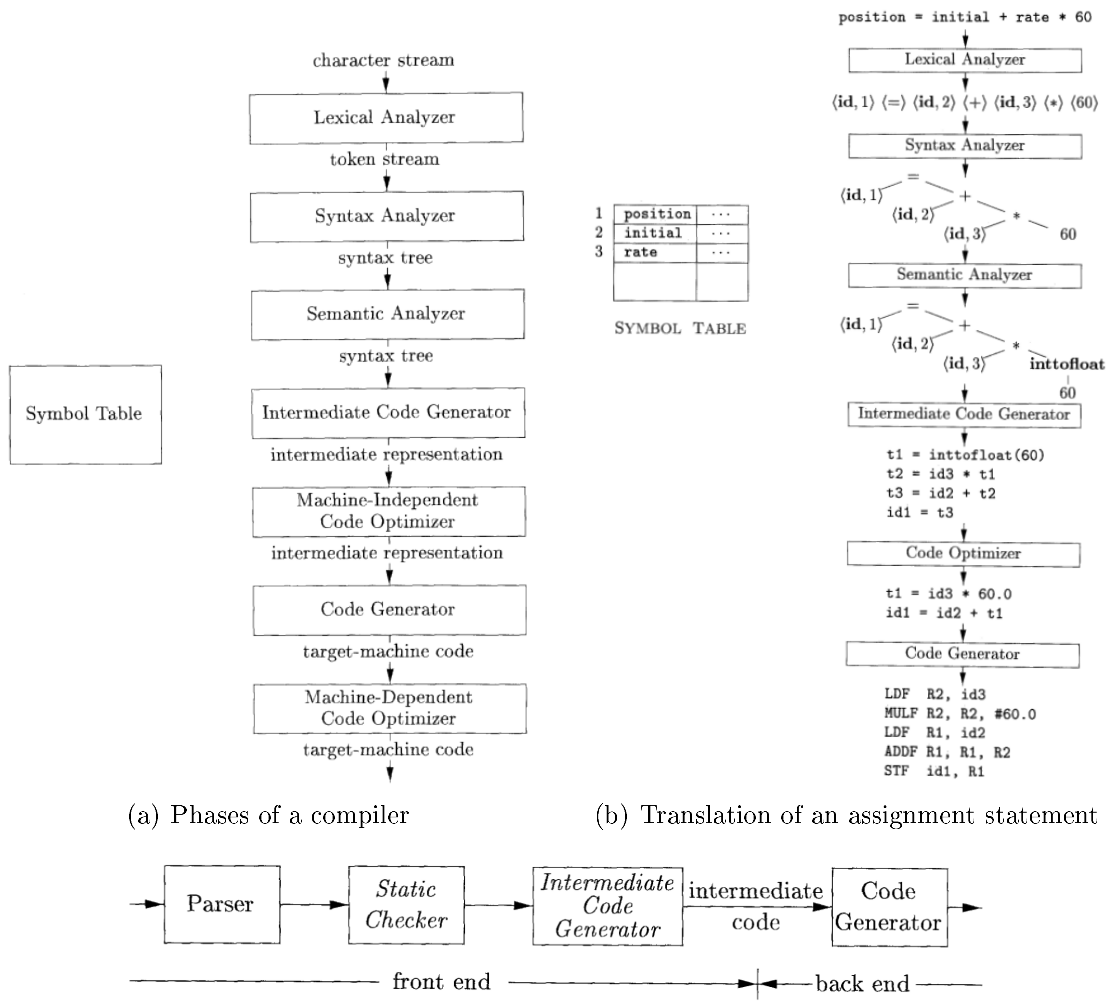
in Figure 2.1(b). This makes the execution of the compiled programs generally very fast, especially when compared to interpreted languages (depicted in Figure 2.1(c)). In this case, an interpreter has to translate each statement to machine code before it can be executed. Languages such as Java employ a two-step compilation process as shown in Figure 2.1(d). Here, the source code is first translated into a machine independent representation which is then converted into the target representation by a *virtual machine*. A benefit of this approach is that it can apply optimizations during the execution of a program.

The design of a compiler is usually based on a pipelined architecture as shown in Figure 2.2(a), resembling the *pipes and filters* pattern commonly known in software design [Bus+96]. In this layout, the compilation process is split into different phases, incrementally transforming the source representation into (optimized) machine code. By using specific intermediate representations, each component can be replaced by a different implementation[1]. According to their position in the pipeline, these steps can be grouped into a language specific compiler *front end* and a (machine specific) *back end*[2].

A concrete example of the processing of a language expression can be seen in Figure 2.2(b). In the first phase of the compiler front end, a *lexical analyzer* (or scanner) breaks down the source code into a stream of tokens that represent the basic constructs of the programming language, i.e. keywords and identifiers. The *syntax analyzer* (or parser) then assembles a syntax tree from these tokens and simultaneously validates the syntactical correctness of the expression. The syntax tree reflects the language structure defined via a context-free grammar. However, even if the syntax of the source code is correct, it may still contain erroneous statements.

[1] The compiler architectures of the GNU Compiler Collection [GS04] and LLVM (http://llvm.org) make use of this method to support a variety of languages and target platforms.

[2] Information that is shared across multiple phases is usually managed in a *symbol table*.


character stream
Lexical Analyzer
token stream
Syntax Analyzer
syntax tree
Semantic Analyzer
syntax tree
Intermediate Code Generator
intermediate representation
Machine-Independent Code Optimizer
intermediate representation
Code Generator
target-machine code
Machine-Dependent Code Optimizer
target-machine code
Symbol Table
SYMBOL TABLE
position = initial + rate * 60
⟨id, 1⟩ ⟨=⟩ ⟨id, 2⟩ ⟨+⟩ ⟨id, 3⟩ ⟨*⟩ ⟨60⟩
inttofloat
Parser
Static Checker
Intermediate Code Generator
intermediate code
front end
back end

1	position	···
2	initial	···
3	rate	···

```
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```

```
t1 = id3 * 60.0
id1 = id2 + t1
```

```
LDF  R2, id3
MULF R2, R2, #60.0
LDF  R1, id2
ADDF R1, R1, R2
STF  id1, R1
```

(a) Phases of a compiler

(b) Translation of an assignment statement

(c) Compiler front end and back end

Figure 2.2.: The structure of a compiler [Aho+06].

For this reason, a *semantic analyzer* performs an additional static validation. This step is often realized using so-called attribute grammars which are extensions of the language's original context-free grammar. Afterwards, the syntax tree is transformed into a machine-independent representation, e.g. Three-Address Code (TAC) or the Static Single-Assignment (SSA) form. In the final step of the frontend, optimization routines - usually implemented as data-flow analyses - are applied to the generated set of instructions.

Once the intermediate representation has been fully processed, it can be converted into instructions for the target system architecture by the compiler back end. Again, an optimization step may be carried out, this time taking into account the specific properties of the system for which the code is being generated.

The relevant phases in the context of this thesis comprise the semantic analysis of the syntax tree and the machine-independent code optimization phase as they implement concepts and methods for static program analysis and thus provide the foundation for a flow-based model analysis.

2.1.2. Formal Languages

In the context of compiler construction, context-free languages play an important role in the definition of programming languages. This class of formal languages[3] originates from Chomsky's effort to devise a rigorous framework for the specification and categorization of text-based languages according to their properties [Cho56]. The reason for the widespread use of context-free grammars can be found in the fact that they provide the essential facilities required for the definition of programming languages while, at the same time, expressions can be analyzed very efficiently. The analysis of the structural composition of a language expression is called *parsing* and the resulting data structure is referred to as a *parse* or *syntax tree*.

A context-free grammar is usually defined in the following way:

Definition 2.1.1

A context-free grammar $\mathcal{G} = (\mathcal{N}, \mathcal{T}, \mathcal{P}, S)$ is a 4-tuple. It consists of finite sets of nonterminals $\mathcal{N}$ and terminals $\mathcal{T}$, productions $\mathcal{P}$ and a start symbol $S \in \mathcal{N}$.

- *Nonterminals or variables with $\mathcal{N} \cap \mathcal{T} = \emptyset$ are sub-languages of the language defined by $\mathcal{G}$. They form the basis for the application of production rules.*
- *The start symbol $S \in \mathcal{N}$ is the nonterminal representing the language of $\mathcal{G}$.*
- *Terminals represent the language's alphabet (i.e. the actual input strings).*
- *Productions $X_0 \rightarrow X_1 ... X_n$ with $X_0 \in \mathcal{N}$ und $X_i \in \mathcal{N} \cup \mathcal{T}, 1 \leq i \leq n$ are rules that are (recursively) applied until all nonterminals have been replaced by terminals.*

A grammar can be expressed using the Extended Backus-Naur Form (EBNF) which was introduced by Wirth [Wir77] as a meta syntax for the formalization of the syntax of the Pascal programming language [JW91]. The official EBNF standard [ISO96] proposes a representation that consists of the following statements[4]:

[3]Exhaustive descriptions of all classes of formal languages, their properties and the relationships to automata and computational complexity theory can be found in [HMU79; Sal87; Lin11; HU69]

[4]In practice, the concatenation operator is often omitted. The introduction of additional helper statements often allows to significantly simplify the definition of a grammar.

Basic statements		**Helper statements**	
Definition	=	Alternate Separator	|
Concatenation	,	Group	( )
Terminal Quotes	" "	Repetition Group	{ }
Rule Terminator	;	Optional Group	[ ]
		Repetition Symbol	*

An important property of the EBNF is its ability to reflectively describe its own syntax [Wir63]:

```
syntax = {statement};
statement = identifier "=" expression ";";
expression = term {"|" term};
term = factor {factor};
factor = identifier | string | "(" expression ")" |
  "[" expression "]" | "{" expression "}";
```

Definition 2.1.2

A language $\mathcal{L}(\mathcal{G})$ generated by a grammar $\mathcal{G}$ is a subset of the possible concatenations of elements of the underlying alphabet: $\mathcal{L} \subseteq \mathcal{T}^*$. A language $\mathcal{L}$ is context-free if and only if there exists a context-free grammar $\mathcal{G}$ with $\mathcal{L} = \mathcal{L}(\mathcal{G})$.

Valid expressions (or words) $w \in \mathcal{L}$ can be generated by (repeatedly) applying production rules to the start symbol S until only terminal symbols are left. Correspondingly, parsers are able to construct derivation trees for given expressions.

2.1.3. Program Representations

We will now shortly discuss two fundamental data structures used by compilers, syntax trees and control-flow graphs. Syntax trees conform to the output of the syntactic phase and the input of the semantic phase while the control-flow graph representation is mainly employed during the optimization step.

2.1.3.1. The Syntax Tree

Based on a context-free grammar $\mathcal{G}$, a parser can be constructed[5] that accepts words (or expressions) $w \in \mathcal{L}(\mathcal{G})$. A parser operates on the token stream generated by the lexical analyzer which consists of terminal symbols and constructs a derivation tree that reflects the application of production rules. The resulting data structure therefore contains a hierarchical ordering of the derivations which have to be applied to the start symbol S to yield w. The inner nodes of this tree, commonly called a *concrete syntax tree* or *parse tree*, conform to the nonterminals on the left hand side of applied production rules while their children represent the (non)terminals on the right hand side.

[5] For example using parser generators such as Yacc [Joh75] or ANTLR [Par07].

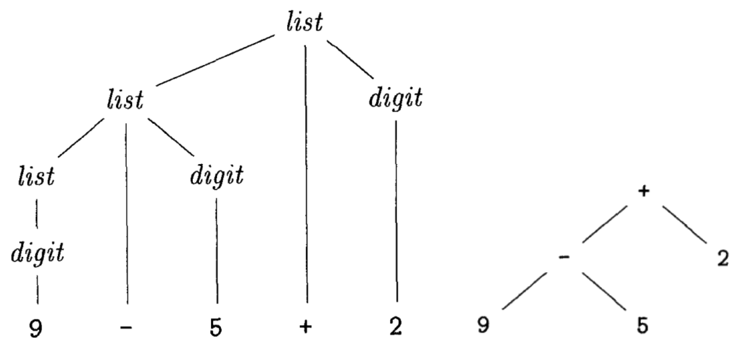
Definition 2.1.3 (*[Aho+06]*)

A parse tree for a context-free grammar $\mathcal{G} = (\mathcal{N}, \mathcal{T}, \mathcal{P}, S)$ is defined as follows:

- *The root is labeled by the start symbol S.*
- *Each leaf is labeled by a terminal $X \in \mathcal{T}$ or by the empty word ϵ.*
- *Each interior node is labeled by a nonterminal $X \in \mathcal{N}$.*
- *If X_0 is the nonterminal labeling some interior node and $X_1, \ldots, X_n$ are the labels of the children of that node from left to right, then there must be a production $X_0 \rightarrow X_1, X_2, ..., X_n$. Here, $X_1, \ldots, X_n$ each stand for a symbol that is either a terminal or a nonterminal.*

  As a special case, if $X \rightarrow \epsilon$ is a production, then a node labeled X may have a single child labeled ϵ.

Since the parse tree directly reflects the syntactic structure of the language expression, it is generally very verbose. For this reason, in practice, this representation is stripped of irrelevant information such as derivation subtrees that are only required for a correct recognition of the structural composition. The result of this simplification process is termed *(abstract) syntax tree*. In this format, inner nodes conform to the semantically relevant constructs of the programming language.



(a) Parse tree (b) Syntax tree

Figure 2.3.: Parse and syntax trees for the expression $9 - 5 + 2$ [Aho+06].

The distinction between the concrete and the abstract syntax can be clarified using the following example which encodes a simple language for arithmetic expressions [Aho+06]:

```
list = list "+" digit | list "-" digit | digit
digit = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Figure 2.3(a) shows the constructed parse tree for the term $9 - 5 + 2$. Its nodes represent the hierarchical application of the production rules. The original input can be obtained by reading the terminal strings at the leaves from left to right.

Removing derivation subtrees for terminal symbols and labeling the inner nodes with their respective tokens leads to the abstract syntax tree depicted in Figure 2.3(b).

The process of creating correct and efficient parsers for language expressions based on a given grammar is a challenging yet well-researched task. However, because this is not in the focus of this thesis, we refer to the canonical literature for more information on this topic.

2.1.3.2. The Control-flow Graph

Control-flow graphs are an alternative representation of the structural composition of language expressions. As such, they are a crucial tool for the application of static optimization techniques. In this section, we outline the basic properties of this data structure and present the graph theoretic properties that are of relevance in the context of data-flow analysis.

As [Aho+06] states, *"the execution of a program can be viewed as a series of transformations of the program state, which consists of the values of all the variables in the program. Each execution of an intermediate-code statement transforms an input state to a new output state. The input state is associated with the program point before the statement and the output state is associated with the program point after the statement."*

The behavioral properties of a program are therefore given by the set of all of its execution paths, i.e. the valid sequences in which its statements can be executed. Static analysis of program instructions must therefore consider all possible execution paths in which a specific statement may be invoked. A control-flow graph is a simple and concise representation from which these paths can be derived.

In the literature, the data structures used to encode the control-flow often differ in certain details. However, their essential properties can be summed up as follows:

Definition 2.1.4

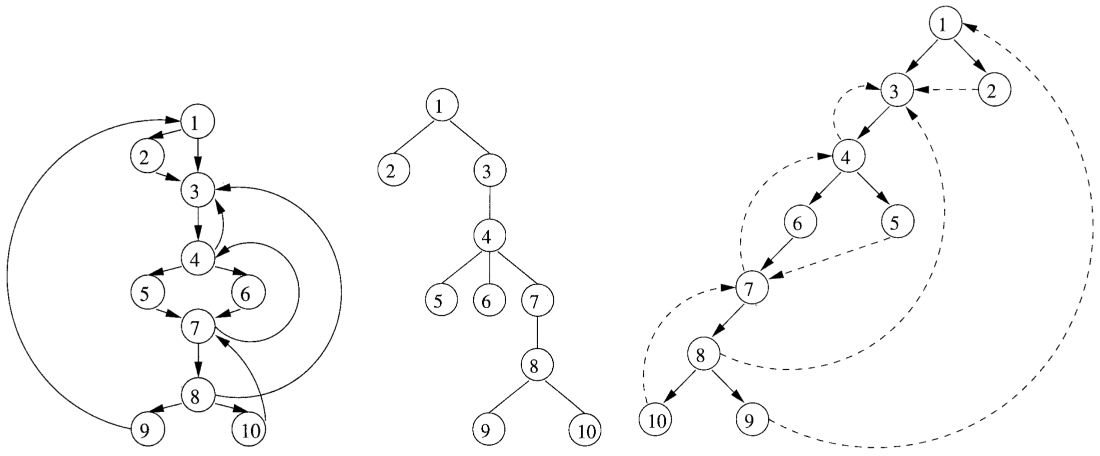
A control-flow graph of a program is a directed graph $CFG = (\mathcal{V}, \mathcal{E}, n_{in}, n_{out})$ with a set of vertices (or nodes) $v \in \mathcal{V}$, a set of directed edges $e = (\mathcal{V} \times \mathcal{V}) \in \mathcal{E}$ with labellings that denote conditional properties and designated entry and exit nodes $n_{in}, n_{out} \in \mathcal{V}$. Nodes usually represent basic blocks[a]. while the predecessor/successor relationships defined by the edges indicate the flow of control, i.e. the execution order[b].

[a]Basic blocks can be derived from the intermediate representation by identifying maximal instruction sequences with a unique entry and exit point (cf. [All70]).
[b]An example of this can be found in Appendix A.1.

The technique of data-flow analysis relies on the computation of information at the control-flow graph's nodes and the propagation of these values along the graph's edges. In the absence of cycles - assuming a correct visiting order - the final result can therefore be calculated in a single pass. However, loops in the control-flow lead to an infinite amount of possible paths which in turn can be of infinite length.

DFA handles this case by applying fixed-point evaluation semantics, i.e. an iterative computation of the result that starts with a defined initialization value.

Since loops are a very common element and often have a substantial impact on the speed of fixed-point convergence, much effort has gone into the study of the properties of cyclic structures. Relevant topics in this context include dominators, depth-first representation, graph reducibility and natural loops.



(a) Original control-flow graph (b) Dominator tree (c) Depth-first representation

Figure 2.4.: Different representations of a control-flow graph [Aho+06].

Dominators The concept of dominators[6] has been introduced by [Pro59]. It is useful in the description of control-flow structures since it reflects properties of the predecessor relationships in execution paths and can therefore be used to derive information about dependencies between nodes.

Definition 2.1.5 (*[LT79]*)

Let $CFG = (\mathcal{V}, \mathcal{E}, r)$ be a flow graph with start vertex r.

A vertex v dominates another vertex $w \neq v$ in CFG if every path from r to w contains v. Vertex v is the immediate dominator of w, denoted $v = idom(w)$, if v dominates w and every other dominator of w dominates v.

Every vertex of a flow graph $G = (\mathcal{V}, \mathcal{E}, r)$ except r has a unique immediate dominator. The edges $\{(idom(w), w) | w \in \mathcal{V} - \{r\}\}$ form a directed tree rooted at r, called the dominator tree of CFG, such that v dominates w if and only if v is a proper ancestor of w in the dominator tree.

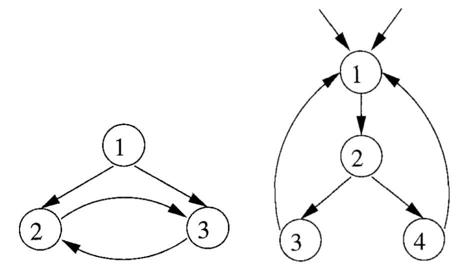
The dominator tree for the graph shown in Figure 2.4(a) is depicted in Figure 2.4(b). In this example, the immediate dominator of node ⑦ is node ④, but not nodes ⑤ and ⑥ since they are "optional" in the sense that they don't lie on *every* path that starts at the entry node.

[6]The computation of dominators can itself be implemented as a data-flow analysis [CHK01].

Depth-first Spanning Tree The depth-first spanning tree (DFST) is another representation of the inherent structural properties of control-flow graphs [HU72]. A spanning tree reflects the order in which nodes are visited when executing a depth-first search on the graph. Back edges resulting from cyclic paths are often omitted. Because the order in which child nodes are visited is arbitrary, it is possible for one control-flow graph to possess multiple depth-first spanning trees. One possible spanning tree for the graph in Figure 2.4(a) is depicted in Figure 2.4(c) with the back edges visualized as dashed lines.

Edges and Reducibility Constructing a depth-first spanning tree yields three different categories of edges:

- *advancing edges* go from a node to one of its successors
- *cross edges* are edges between nodes that do not have an ancestor relationship.
- *retreating edges* go from a node to one of its predecessors (or the node itself)



(a) Canonical nonreducible flow graph (b) Two loops with single header

Figure 2.5.: Special cases of cyclic paths [Aho+06].

In most cases, *retreating edges* in the depth-first spanning tree denote *back edges* in the control-flow graph. An edge is termed back edge if its head dominates its tail. If all retreating edges in a graph are also back edges, it is called *reducible*, otherwise it is *nonreducible*. The latter case implies that the graph will still be cyclic even after the removal of all back edges. The canonical example shown in Figure 2.5(a) has two different DFST representations with $2 \to 3$ and $3 \to 2$ being retreating but not back edges.

Natural Loops Programs often spend a lot of their execution time running loops. It is therefore desirable to study their properties. So-called *natural loops* - loops with only one entry point - are easier to handle since flow analysis can determine certain properties that hold for each run. Therefore it is important to be able to distinguish this class of loops from others which do not possess this property. Natural loops are characterized by a *loop header*, the single entry node, and a back edge to the header. The natural loop of a back edge consists of the header node itself and all nodes which can reach the source of the back edge without going through the header. As an example, for the graph shown in Figure 2.4(a), the loop of the back edge $10 \to 7$ is $\{7, 8, 10\}$. The loop for $4 \to 3$ is $\{3, 4, 5, 6, 7, 8, 10\}$ since the "succeeding" loops can return to 4 without going through 3. The loop for $10 \to 7$ is nested in the $4 \to 3$ loop.

If a graph is reducible, i.e. every retreating edge is a back edge, a natural loop can be assigned to each retreating edge. An important property of natural loops is that they are either disjoint or nested in each other, imposing a hierarchical structure. There is however an exception: If, like in the case depicted in Figure 2.5(b), two loops share the same header, they must be merged into a single loop. In this example, the natural loops of the back edges $3 \to 1 : \{1, 2, 3\}$ and $4 \to 1 : \{1, 2, 4\}$ would be combined to $\{1, 2, 3, 4\}$.

2.1.4. Program Analysis

As [NNH99] notes, *"[p]rogram analysis offers static compile-time techniques for predicting safe and computable approximations to the set of values or behaviours arising dynamically at run-time when executing a program"* with the goal of transforming *"the program (at the source level or at some intermediate level) inside a compiler so as to obtain better performance"*. Additional use cases listed in [KSK09] include determining the validity of a program and facilitating its execution, e.g. through dynamic type inferencing.

The approximation of program behavior is required since *"mathematically, the problem of generating an optimal target program for a given source program is undecidable"* [Aho+06]. As a consequence, [NNH99] states that *"[o]ne common theme behind all approaches to program analysis is that in order to remain computable one can only provide approximate answers"*. Therefore, the possibilities for static optimization are inherently limited by the fact that a conservative approach will typically lead to an over approximation of the actual behavioral properties.

Over time, many techniques have been developed for the purpose of program analysis. In addition to data-flow analysis, [KSK09] lists inference and constraint resolution systems, model checking and abstract interpretation as examples. While each of these methods employs its own theoretical abstractions and has its unique application scenarios, they all share some common characteristics with respect to the topic of program analysis. To categorize the scope and inherent properties of the DFA approach along these lines, we include a short summary of the classification system proposed by [KSK09]:

Time of Performing Analysis Analysis performed on the information that is available at design time is termed *static analysis* while an analysis that depends on information which only becomes available after execution is named *dynamic analysis*.

Scope of Analysis Since programs are usually structured hierarchically, the scope of the analysis can be anything from a single statement to the whole program. Usually, the analysis of a structure requires the preceding analysis of its substructures.

Flow Sensitivity of Analysis If the result calculated for a specific program point depends on the paths leading from/to this point, the analysis is *flow sensitive*, otherwise it is *flow insensitive*.

Context Sensitivity of Analysis If an analysis takes different calling contexts into account then it is *context sensitive*, if the information is calculated in a way that holds for all calling contexts it is *context insensitive*.

Granularity of Performing Analysis While *incremental analysis* is able to incorporate changes into the results of a previous analysis run, an *exhaustive analysis* always starts from scratch.

Program Representations Used for Analysis Typical representations used for analysis are sequences of instructions, trees or graphs.

Representations of Information Information is usually represented in the form of (finite) sets of valid model states, facts or program entities.

2.2. Attribute Grammars

The abstract syntax of a programming language is commonly given in the form of a context-free grammar. However, due to the nature of this formalism, some restrictions on the structure of valid language expressions cannot be encoded in the grammars themselves (cf. Section 2.1.1). This includes, for example, validations of operators in mathematical expressions to ensure that their datatypes are compatible with respect to the applied operation.

These types of validations are carried out in the semantic analysis phase of the compilation process. The most important technique that is employed for this purpose are attribute grammars, an extension of a programming language's defining grammar with semantic attributes. These attributes are assigned to symbols in the grammatical productions. Their results are computed by semantic rules that derive the value of a specific attribute from values of other attributes in the same production. Attributes, as well as semantic rules, can be instantiated for parsed language expressions. This results in an information flow that transports values either in a top-down or a bottom-up direction in the syntax tree, thereby providing each subexpression with contextual information computed at its respective predecessors or successors. Since an attribute grammar defines only the structural properties of an analysis but not the evaluation logic, it constitutes a declarative specification.

While this approach bears some resemblance to the data-flow analysis technique, it possesses several distinctive features: For one, attribute grammars allow a very efficient evaluation that can be carried out in a single pass and even be integrated with the syntax analysis phase. As such, this method does not provide support for fixed-point evaluation semantics. Secondly, by their very nature, attribute grammars are tightly integrated with the definition of the target language, i.e. the analysis specification and evaluation methodology makes heavy use of the structural composition of the language expressions.

The approach that is the subject of this thesis is partly inspired by this technique as it employs a comparable attribution-based method for annotating flow analysis specifications at metamodels. For this reason, we will now provide a short description of the basic notions and the most important properties of attribute grammars: Section 2.2.1 presents the attribute grammar formalism and some of its associated concepts while Section 2.2.2 describes how attribute-based analyses can be instantiated and evaluated for arbitrary language expressions.

2.2.1. Specification of Attribute Grammars

The attribute grammar approach, first proposed by [Knu68], is a topic that is widely discussed in the canonical compiler construction literature. It can be observed that the properties of this formalism as well as the presented use cases often differ with respect to some details. The descriptions in this section aim to provide a unified understanding of this concept by incorporating information from different sources such as [Aho+06; WM95; WG98; Kas90; Thi09].

2.2.1.1. The Attribute Grammar Formalism

An attribute grammar AG represents an extension of a traditional context-free grammar $\mathcal{G} = (\mathcal{N}, \mathcal{T}, \mathcal{P}, S)$, consisting of nonterminals $\mathcal{N}$, terminals $\mathcal{T}$, production rules $\mathcal{P}$ and a start symbol S (cf. Section 2.1.2).

The specification of an attribute grammar associates semantic attributes, which are either of the type *synthesized* or *inherited*, with the (non)terminals of the underlying grammar. The assignment of an attribute A to a grammatical symbol $X = \mathcal{N} \cup \mathcal{T}$ implies the presence of occurrences of A at each occurrence of X inside one of the grammar's production rules $P \in \mathcal{P}$. In this sense, the specification of a semantic attribute, often consisting of a name and a datatype, conforms to a global type definition that is shared by all of its occurrences. For each production rule containing attributes, one or more semantic rules have to be specified that describe how the attributes can be evaluated. Depending on the attribute type, an attribute occurrence may either constitute an input parameter of a semantic rule or represent the target to which the result value is assigned. Synthesized attributes transfer values from the right hand side of the production to the left hand side while the information flow for inherited attributes follows the opposite direction.

The parsing of a language expression yields a syntax tree (cf. Section 2.1.3). This data structure describes the transformations that have to be applied to the grammar's start symbol to arrive at the target expression. Each selection of neighboring nodes in the syntax tree therefore represents the application of a specific production rule. Consequently, attribute occurrences located at the (non)terminals inside a production rule can be instantiated for all applications of the respective production rule in the syntax tree. Based on the definition of inherited and synthesized attributes, this results in an information flow that transports values either in a top-down or a bottom-up direction. The instantiation and computation of instantiated attributions is further discussed in Section 2.2.2.

We now present a definition of attribute grammars that includes the central properties of this approach:

Definition 2.2.1

An attribute grammar is a 3-tuple $AG = (\mathcal{G}, \mathcal{A}, \mathcal{R})$ consisting of

- *a context-free grammar $\mathcal{G} = (\mathcal{N}, \mathcal{T}, \mathcal{P}, S)$.*
- *a finite set of attributes $\mathcal{A}$ where $A(X) \subseteq \mathcal{A}$ denotes attributes associated with a grammatical symbol $X \in \mathcal{N} \cup \mathcal{T}$.*
- *a finite set of semantic rules $\mathcal{R}$. A semantic rule $R(P) = \{X_i.a \leftarrow f(X_j.b, \dots, X_k.c)\}$ computes the result for the attribute $a \in A(X_i)$ based on the values of $b \in A(X_j), c \in A(X_k), \dots$ for attributes located at (non)terminals $X_i, X_j, X_k, \dots \in P$ using a function f.*

2.2.1.2. Synthesized and Inherited Attributes

$A(X)$ can be subdivided into two disjunct sets $inh(X)$ and $syn(X)$, containing inherited and synthesized attributes respectively: $A(X) = inh(X) \cup syn(X)$ and $inh(X) \cap syn(X) = \emptyset$. The sets of all inherited and synthesized attributes can therefore be defined as $inh = \bigcup_{X \in \mathcal{N} \cup \mathcal{T}} inh(X)$ and $syn = \bigcup_{X \in \mathcal{N} \cup \mathcal{T}} syn(X)$.

Synthesized attributes propagate results in an upwards direction in the syntax tree. This is required when results for composite expressions depend on the results of their partial expressions. An example use case would be the evaluation of an arithmetic expression which has to be preceded by an evaluation of its partial expressions. Inherited attributes, on the other hand, transport values from the top to the bottom of the syntax tree. This can be useful if contextual information, such as the set of currently defined variables, has to be made available to subexpressions.

[Aho+06] provides the following definitions for the two attribute types:

Synthesized Attributes

A synthesized attribute for a nonterminal A at a parse-tree node N is defined by a semantic rule associated with the production at N. Note that the production must have A as its head. A synthesized attribute at node N is defined only in terms of attribute values at the children of N and at N itself.

Inherited Attributes

An inherited attribute for a nonterminal B at a parse-tree node N is defined by a semantic rule associated with the production at the parent of N. Note that the production must have B as a symbol in its body. An inherited attribute at node N is defined only in terms of attribute values at N's parent, N itself, and N's siblings.

2.2.1.3. Notation

To enable a concise definition and to support an automatic evaluation of attribute grammars, a specification language, i.e. a concrete syntax, is required. However, in the field of compiler construction, there is no general consensus on this topic and thus no universal syntax exits. Consequently, tools such as ANTLR [Par07] and Yacc [Joh75] often define a proprietary format. Nevertheless, since all attribute grammar frameworks share the same properties, it is possible to list the features that an appropriate specification language has to implement. Generally speaking, an attribution language must support the definition of attributes and semantic rules and additionally support the association of these constructs with their respective counterparts (symbols and productions) in the underlying grammar.

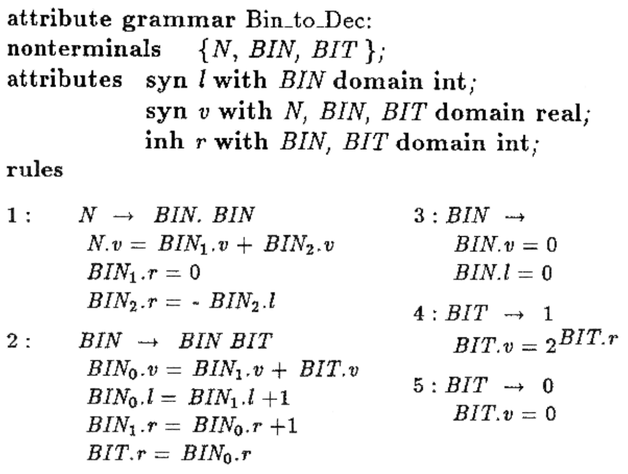
Depending on the respective framework, attributes can be assigned a value domain, i.e. a datatype. Additionally, they must be classified as either inherited or synthesized. Finally, each attribute requires the specification of a set of grammatical symbols which should be annotated with occurrences of the attribute. Semantic rules can be written as equations that are defined in the context of a specific production rule. The attributes which have occurrences in the respective production are often addressed as $X.a$ where X is an occurrence of a grammatical symbol and $a \in A(X)$ is an occurrence of an attribute which has been assigned to this symbol. If a (non)terminal X has multiple occurrences in the same production P, an index must be used to denote the correct element.

It should be noted that, in order to ensure their well-formedness, attribute grammars must themselves be subjected to a semantic analysis. For example, each attribute has to belong to either *inh* or *syn*. Also, all attribute occurrences required as input by a semantic rule must in fact be present in the respective production: $\forall R \in \mathcal{R} : X.a \in R(P) \rightarrow X \in P \wedge a \in A(X)$.

Since attribute grammars represent an extension of traditional context-free grammars, a possible way to specify the syntax of an attribution language consists of an extension of the meta language EBNF. [WM95] proposes a notation that combines the representation of a context-free grammar with the definition of semantic attributes (see below).

2.2.1.4. Example

We will now illustrate the presented principles in the context of an attribute grammar called *Bin_to_Dec* (cf. [WM95]). The attribute grammar which is depicted in Figure 2.6 converts binary numbers in the form $n.m$ with $n, m \in \mathbb{N}$ to a decimal representation. The employed notation indexes the occurrences of the nonterminal *BIN* with their position in the grammatical production rule.



attribute grammar Bin_to_Dec:
nonterminals $\{N, BIN, BIT\}$;
attributes **syn** l **with** BIN **domain int**;
syn v **with** N, BIN, BIT **domain real**;
inh r **with** BIN, BIT **domain int**;
rules

1 : $N \rightarrow BIN.\ BIN$
$N.v = BIN_1.v + BIN_2.v$
$BIN_1.r = 0$
$BIN_2.r = -\ BIN_2.l$

2 : $BIN \rightarrow BIN\ BIT$
$BIN_0.v = BIN_1.v + BIT.v$
$BIN_0.l = BIN_1.l + 1$
$BIN_1.r = BIN_0.r + 1$
$BIT.r = BIN_0.r$

3 : $BIN \rightarrow$
$BIN.v = 0$
$BIN.l = 0$

4 : $BIT \rightarrow 1$
$BIT.v = 2^{BIT.r}$

5 : $BIT \rightarrow 0$
$BIT.v = 0$

Figure 2.6.: Attribute grammar *Bin_ to_ Dec* [WM95].

This process employs the following attributes:

v (*value*, *synthesized*)
: Computes the decimal value of the binary digits depending on their rank r and synthesizes the composite value.

l (*length*, *synthesized*)
: Computes the amount of digits after the decimal point (required for r).

r (*rank*, *inherited*)
: Computes the rank of each binary digit. Elements on the left hand side of the decimal point possess ascending ranks $0, 1, 2, \ldots$ while elements on the right hand side have descending ranks $-1, -2, -3, \ldots$

This example illustrates the combination of synthesized and inherited attributes: First, the length of the binary number after the decimal point is determined and the rank of the single digits is set. The rank of all other binary digits is computed by inheriting and incrementing the rank of the respective predecessor. Finally, the decimal value can be computed for each digit. The results are synthesized to yield the final value at the root of the syntax tree.

The information flow that results from the attributes and the semantic rules is depicted in Figure 2.7. As a convention, inherited attributes are always placed on the left hand side of grammatical symbols while synthesized attributes are annotated on the right hand side. If an attribute occurrence a depends on the value of an occurrence b, this relationships is visualized by an arrow pointing from b to a.

2.2.2. Instantiation and Evaluation of Attribute Grammars

Once an attribution has been defined for a grammar, it can be instantiated and evaluated for arbitrary language expressions. The first step of this process consists of the construction of an attributed syntax tree which associates the tree's nodes with

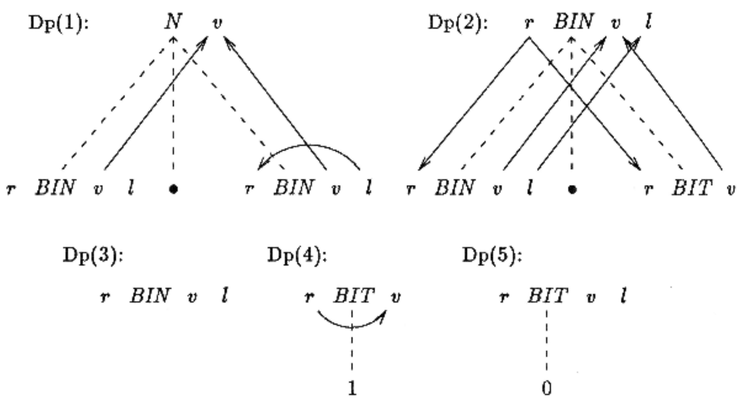
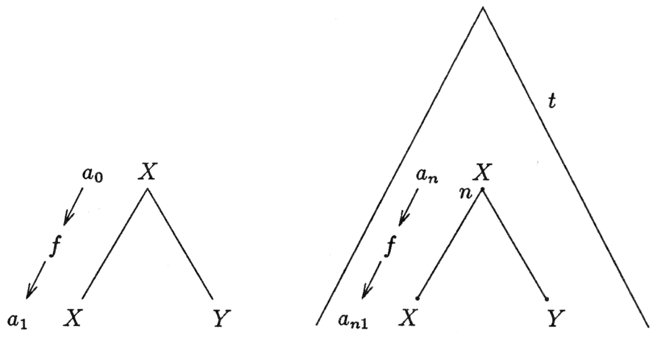


Figure 2.7.: Information flow between attribute occurrences for *Bin_to_Dec* [WM95]

attribute instances. Since attributions are declarative specifications, a dependency analysis must subsequently be carried out to determine a correct evaluation order for the instances. Finally, the result can be computed by executing the semantic rules in the derived order.

2.2.2.1. Attributed Syntax Trees

An attribute grammar is fully defined by the context-free grammar, the semantic attributes, their associations with grammatical symbols and the set of semantic rules. Based on this definition, attribute instances can be created for a parsed language expression, i.e. a syntax tree. Since the tree's nodes represent the (non)terminals of the underlying grammar, the set of attribute instances located at each node conforms to the attributes assigned to the respective symbol. Attribute results can then be stored in the instances' data fields. For performance reasons, the instantiation step is often carried out by the parser during the processing of the language expression.



(a) Attribute occurrences (b) Attribute instances in the syntax tree

Figure 2.8.: Attribute occurrences and attribute instances [WM95].

Figure 2.8 illustrates the difference between an attribute occurrence and an attribute instance. In this case, we assume the existence of a production $X \to XY$

with $X, Y \in \mathcal{N}$, an attribute $a \in inh(X)$ and a semantic rule $X_1.a = f(X_0.a)$ which propagates the value of a downwards.

Figure 2.8(a) shows the dependency graph for the production. The symbol X, which occurs twice in the production rule, possesses an occurrence of a. The semantic rule encodes an information flow from the occurrence on the left hand side of the production rule, designated a_0, to the right hand side occurrence a_1.

We now assume that a syntax tree has been generated by a parser that includes an application of the production rule $X \rightarrow XY$. The remainder of the tree is represented by t. In Figure 2.8(b), it can be seen that the nodes of the n-th application of the production possess instances of the occurrences named a_n and a_{n1}.

2.2.2.2. Dependency Analysis

Once an attributed syntax tree has been created, the attribute instances attached to the nodes in the tree have to be evaluated. The dependency relationships between the instances play an important role in this process since the input dependencies of each rule must be satisfied before it can be executed. Regarding the evaluation order, it can be stated that, if a semantic rule that computes a result for an attribute instance a depends on the value of the instance b, the rule which yields the result for b has to be invoked first. However, since an attribute grammar is a purely declarative specification, it does not provide any specific evaluation logic in and of itself. A dependency analysis step is therefore required to derive a valid execution order that ensures that input values are always made available before they are needed. This can be carried out dynamically by constructing a global dependency graph from the local production dependency graphs (cf. Figure 2.7 and Figure 2.9). Alternatively, special types of attribute grammars can be used which guarantee that dependency relationships always follow an established pattern.

As is the case with data-flow analysis, the sum of all semantic rules applied to the language instance forms an equation system. However, unlike the DFA approach, the case of cyclic dependencies is usually regarded as *undefined*. To ensure that a unique solution exists for any instantiation of an attribute grammar, its specification has to be *well-defined*: *"An attribute grammar is well-defined if, for each structure tree corresponding to a sentence of $\mathcal{L}(\mathcal{G})$, all attributes are effectively computable"* [WG98]. This property can be guaranteed if the grammar adheres to a specific set of restrictions.

2.2.2.3. Evaluation

The evaluation of an attributed syntax tree involves the invocation of the semantic rules and the assignment of result values to the attribute instances located at the tree's nodes. As stated above, it is important that the execution order of the rules respects the dependency relationships between the instances.

According to [WM95], the overall evaluation process can be divided into two main phases:

Strategy phase

A valid evaluation order is derived based on the results of the dependency analysis step.

Evaluation phase

Results for attribute instances are computed by executing rules in the specified order.

It is also possible to carry out an evaluation dynamically. In this case, the dependency relationships are evaluated before/after each semantic rule is executed. Generally, there are two different strategies that follow this pattern:

Demand-driven approach

Starting with a set of instances for which results should be computed, the input dependencies are recursively included in the calculation.

Data-driven approach

Starting with a set of existing results, instances are iteratively computed once all of their input dependencies are available.

Depending on the combination of the methods employed to analyze dependencies and to derive valid evaluation orders, approaches to solve attribute grammar equation systems can be grouped into two categories:

Dynamic approaches

Dynamic approaches don't require any previous knowledge about the structure of the attribution and the resulting equation system. The evaluation order is computed individually for each attributed syntax tree.

Static approaches

Static approaches employ preexisting knowledge about the structure of dependency relationships. This can be used to construct efficient static evaluators for arbitrary syntax trees.

Examples for fully dynamic methods include the demand and data-driven approaches mentioned above. These basically constitute elimination methods which continuously search for attribute instances that can be safely evaluated at any given point in time. However, unsurprisingly, these algorithms are usually not very efficient. In practice, static approaches are therefore preferred. Special categories of attribute grammars - such as L- or S-attributed grammars - exist that impose restrictions on the structure of attributions but at the same time can anticipate the nature of the resulting dependency relationships [Aho+06]. An additional advantage of this approach is that the instantiation of the attributes as well the computation of the results can be carried out during the syntactical analysis of the language expression.

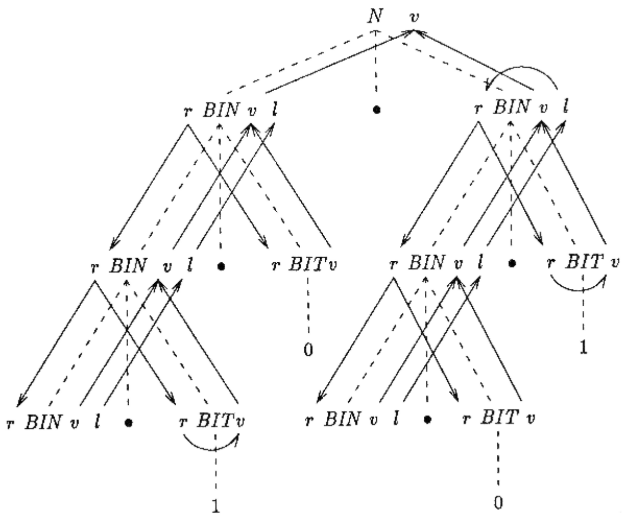


Figure 2.9.: Dependency graph for *Bin_to_Dec* and the expression “10.01” [WM95].

2.2.2.4. Example

The example shown in Figure 2.9 is based on the grammar *Bin_to_Dec* (cf. Figure 2.6). It shows the syntax tree for the expression “10.01” along with the instantiated attributes and their dependency relationships. Based on this information, it is now possible to derive a valid execution order for the semantic rules.

2.3. Data-Flow Analysis

The data-flow analysis approach is commonly employed for the purpose of optimizing programs during the compilation process by associating *“with every program point a data-flow value that represents an abstraction of the set of all possible program states that can be observed for that point”* [Aho+06]. Information that can be computed this way includes, for example, the liveness state of variable assignments inside basic blocks. To preserve the semantics of the original program, results computed for each program point have to hold for all possible executions of the respective program. The evaluation of each statement therefore has to take into account all of its execution paths. Since runtime properties are assessed based on statically available information, this approach is a typical example of a static analysis technique.

The method developed in this thesis leverages the traditional data-flow analysis concept to enable the analysis of models. For this reason, we will now provide an overview of the relevant aspects of conventional flow analysis techniques: In Section 2.3.1, we present a survey of the motivation, history and the basic principles behind this method followed by a formal definition in Section 2.3.2. Canonical algorithms for the solving of data-flow equation systems, namely the round-robin and worklist methods, are described in Section 2.3.3.

2.3.1. History and Conceptual Overview

The technique of data-flow analysis is commonly employed to derive information from a program's control-flow, usually for optimization purposes. [KSK09] states that data-flow analysis *"is a classical static analysis technique that has been used to discover useful properties of programs being analyzed"* with *"applications ranging from compiler optimizations to software engineering to software verification [. . .] to prove the soundness of programs with respect to properties of interest"*.

In the history of compiler construction, code optimization techniques in general, and the DFA approach in particular, have been used from very early on to improve the quality of the generated programs. At the time, no consensus existed with respect to what constitutes a good compiler architecture and reasonable optimization techniques. Therefore, compiler designers were often forced to improvise in order to ensure that programs made best use of the limited capabilities that the available hardware offered. As a result, it can be observed that *"the practice of data flow analysis precedes the theory"* [KSK09]. According to [Hec77], the idea of using DFA dates back as far as 1966 in which an unpublished technical report [VW63] gave a first description of the iterative algorithm used for solving DFA equation systems. The authors of [Aho+06] name Allen [All70] and Cocke [Coc70] as possible candidates for the first scientific exploration of this concept. This view is shared by the authors of [KSK09] who also include Kennedy [Ken71] in this list. Kildall [Kil73], Kam and Ullman [KU76; KU77] are credited with the introduction of theoretical abstractions such as lattice-theory.

The "dragon book" [Aho+06], which is often cited as the standard work in the field of compiler construction, describes data-flow analysis as *"a body of techniques that derive information about the flow of data along program execution paths"*. This is implemented by a schema that *"defines a value at each point in the program. Statements of the program have associated transfer functions that relate the value before the statement to the value after. Statements with more than one predecessor must have their value defined by combining the values at the predecessors, using a meet (or confluence) operator"*. The results can be interpreted in a way that *"for each instruction in the program, they specify some property that must hold every time that instruction is executed"*. In other words, flow analysis yields a static approximation of a program's dynamic properties. An example for this is the *variable liveness analysis*, a common optimization technique that *"determines, for each point in the program, whether the value held by a particular variable at that point is sure to be overwritten before it is read"*.

As has been stated, the evaluation of a DFA equation system yields an approximation of a program's dynamic properties. This is a necessary compromise since, in most cases, programs define an infinite number of execution paths. In the general case, determining exact results is therefore an undecidable problem (cf. the halting problem and Rice's theorem [Ric53]). Since the properties computed by a flow analysis are guaranteed to hold for all possible execution paths, results are considered to be a *conservative* (or *safe*) approximation. Due to the static nature of the analysis, imprecisions therefore cannot be avoided if the original execution semantics are to be preserved. If, for example, a variable definition is overwritten on one alternative path but not on the other, it must be assumed that it will reach subsequent instructions even if this case will never happen during the actual execution of the program. While this premise may lead to “missed opportunities”, it also guarantees that the transformed program exhibits the same behavior as its unoptimized counterpart.

At this point, we include a classification of the data-flow analysis technique along the lines of the framework presented in Section 2.1.4. The following description is a short summary of the categorization provided by [KSK09]:

Time of Performing Analysis

DFA is usually employed for static analysis. Exceptions include methods such as dynamic program slicing which requires information about execution traces [KL88].

Scope of Analysis

Flow analysis can be used across different levels of scope, e.g. within a basic block, confined to a function (*intraprocedural*) and across multiple functions (*interprocedural*).

Flow Sensitivity of Analysis

Since information is propagated along a program's control-flow graph, most use cases are inherently flow-sensitive.

Context Sensitivity of Analysis

Some interprocedural flow analyses take different calling contexts into account. Because of the complexity of this approach and due to the performance impact, this is however only done to a limited degree.

Granularity of Performing Analysis

Both exhaustive and incremental analysis is possible although incremental versions are more difficult to implement [Ryd83].

Program Representations Used for Analysis

Usually, data-flow analysis relies on control-flow graphs, although in some cases, other representations such as abstract syntax trees or the Static Single-Assignment form may also be used.

Representations of Information

The most common result representations are sets of program entities such as variables or expressions satisfying the given property. To improve performance, information is often encoded in bitvectors.

2.3.2. Global Intraprocedural Data-flow Frameworks

It has been discovered that many problems that can be solved through flow analyses share a set of common properties. Any specific problem therefore can be considered to be a member of a larger class of similar problems. In the following, we will present the theoretical foundations of a canonical model for *data-flow frameworks* (based on the definitions from [Aho+06; KSK09]) that supports the specification of global, intraprocedural flow analyses.

2.3.2.1. Transfer Functions

A data-flow analysis typically operates on a program's control-flow graph which can be derived from its machine-independent intermediate representation (cf. Section 2.1.3). Information is propagated from node to node either along the program's control-flow (*forward analysis*) or in the opposite direction (*backward analysis*). At each node, the incoming information is first aggregated and then manipulated to reflect the effects of the local node's instructions. The computation of the data-flow result at a particular node therefore consists of two steps:

Global Data-flow Analysis

First, the program state before the execution of the respective basic block has to be assessed. For this purpose, state information retrieved from all preceeding (or succeeding) nodes must be combined using a *meet operator*.

Local Data-flow Analysis

The result of the first step is then modified to reflect the effects of the basic block. As consecutive instructions may affect the entities, e.g. by accessing the same variables, the block's statements have to be processed one by one.

These steps can be implemented in the form of standardized *transfer functions* which operate on algebraically specified value domains (*meet semilattices*) to ensure a concise, correct and efficient execution. For local data-flow analysis, the static approximation of the state before the execution of a statement s is referred to as $\texttt{IN}(s)$ while the state after the execution of s is given by $\texttt{OUT}(s)$. The relationship between $\texttt{IN}(s)$ and $\texttt{OUT}(s)$ can therefore be modeled as

$$\texttt{OUT}[s] = f_s(\texttt{IN}[s])$$

where the transfer function f_s transforms the input into the corresponding output state. Backward analyses which propagate information in the opposite direction, i.e. compute information that is *upwards exposed*, can be calculated using the reversed definition:

$$\texttt{IN}[s] = f_s(\texttt{OUT}[s])$$

Since, by definition, the control-flow inside basic blocks is always sequential, the overall result for a block can be computed by chaining the block's transfer functions:

$$\texttt{IN}[s_{i+1}] = \texttt{OUT}[s_i]$$

As transfer functions are responsible for the propagation of information from one point to another, their implementation usually follows a common pattern. In practice, it can be assumed that the effects of a single instruction (and consequently the effect of whole basic blocks) consist of

1. the removal of information which will not be available thereafter and
2. the adding of information that will be visible (*downwards* or *upwards exposed*) in subsequent parts of the control-flow.

The removal of information is carried out by a function named $kill_s$ which returns a set of elements which are "destroyed" by the respective statement s. Similarly, gen_s denotes newly generated information. As an example, we can consider a variable assignment statement that destroys the previous assignment to the target variable but at the same time results in a new assignment to an expression or a concrete value. Each transfer function applied to a value x is therefore expected to remove a set of elements while adding others. Generally, transfer functions can be written as:

$$f_s(x) = gen_s \cup (x - kill_s)$$

Transfer functions applied to the statements $s_1, s_2, ...s_n$ inside block B thus implement the local flow analysis. The same principle however also applies to global analysis, i.e. the propagation of information between basic blocks. In this context, the "borders" of a block B, i.e. its first and the last statement (s_1 and s_n), represent the connection points between the local and the global analysis:

$$\begin{aligned} \texttt{IN}[B] &= \texttt{IN}[s_1] \\ \texttt{OUT}[B] &= \texttt{OUT}[s_n] \end{aligned}$$

The transfer function f_B for a block B can therefore be defined as the composition of the statement transfer functions:

$$f_B = f_{s_n} \circ ... \circ f_{s_2} \circ f_{s_1}$$

The major difference between the local and the global analysis can be found in the fact that the control-flow inside a block is always sequential while the global control-flow usually includes branches which denote different execution paths. As a program point may therefore be reached in a number of different ways, the computation of a conservative approximation requires a problem-specific *meet operator* $\wedge \in \{\cup, \cap\}$ which combines the results from preceeding/succeeding blocks.

Consequently, `IN` and `OUT` values for basic blocks are specified as follows[7]:

$$\begin{aligned} \texttt{IN}[B] &= \bigwedge_{P \in predecessors(B)} \texttt{OUT}[P] \\ \texttt{OUT}[B] &= f_B(\texttt{IN}[B]) \end{aligned}$$

[7]Again, the functions can be reversed to perform a backward analysis.

In the case of detecting unused variables, the set union $\cup$ needs to be chosen as meet operator $\wedge$, since it must be assumed that the execution of a block is preceded by the execution of any of its predecessors. A safe approximation therefore has to compute the union of the variables that reach a block on *any* incoming path and discard these elements from the set of unused variables. [KSK09] states that *"the all-path variant of data flow information is also called must information. Analogously, the any-path variant of data flow information is called may information"*.

2.3.2.2. Value Domains

The process of solving a DFA equation system requires the computation of the `IN` and `OUT` values. To ensure that the analysis terminates in a unique fixed-point solution, it is necessary to impose some restrictions on the value domains. Typically, the properties of data-flow result sets as well as their behavior with respect to the application of the meet operator are defined using algebraic specifications.

The (meet) *semilattice*[8] lends itself as an appropriate structure since it not only imposes an order on the value domain but also defines a *top* element. This property ensures the algorithm terminates after a finite number of function applications.

Definition 2.3.1 (*[Aho+06]*)

A meet semilattice $(\mathcal{V}, \wedge)$ consists of a set of values $\mathcal{V}$ and a binary meet operator $\wedge$ such that for all x, y, $z \in \mathcal{V}$:

1. *$x \wedge x = x$ (meet is idempotent).*
2. *$x \wedge x = y \wedge x$ (meet is commutative).*
3. *$x \wedge (y \wedge z) = (x \wedge y) \wedge z$ (meet is associative).*

A semilattice has a top element, denoted $\top$, such that

$$\forall x \in \mathcal{V},\, \top \wedge x = x$$

Effectively, $\top$ therefore acts as a neutral element with respect to the $\wedge$ operator. Optionally, a semilattice may have a bottom element, denoted $\bot$, such that

$$\forall \in \mathcal{V},\, \bot \wedge x = \bot$$

The meet operator has to define a *partial order* $\leq$ on elements $v \in \mathcal{V}$. *"In the context of data flow analysis, the relation $\leq$ can be interpreted as a conservative (safe) approximation [. . .]: If $x \leq y$, then, in any context, the data flow value x can be used in place of y for optimization without affecting the correctness of the optimized program"* [KSK09].

[8] In contrast to a lattice, the meet semilattice does not define an additional join operator $\vee$. According to [Aho+06], the canonical data-flow literature focuses on meet semilattices although it would be possible to use semilattices with a join operator instead.

Definition 2.3.2 ([Aho+06])

A relation $\leq$ is a partial order on a set $\mathcal{V}$ if for all x, y, $z \in \mathcal{V}$:

1. *$x \leq x$ (the partial order is reflexive).*
2. *If $x \leq y$ and $y \leq x$, then $x = y$ (the partial order is antisymmetric).*
3. *If $x \leq y$ and $y \leq z$, then $x \leq z$ (the partial order is transitive).*

The pair $(\mathcal{V}, \leq)$ is called a poset, or partially ordered set.
It is also often convenient to have a $<$ relation for a poset, defined as

$$x < y \text{ if and only if } (x \leq y) \text{ and } (x \neq y).$$

Now, it is possible to define a partial order $\leq$ for a semilattice $(\mathcal{V}, \wedge)$ as follows:

Definition 2.3.3 ([Aho+06])

For all $x, y \in \mathcal{V}$, we define

$$x \leq y \text{ if and only if } x \wedge y = x.$$

Because the meet operator $\wedge$ is idempotent, commutative, and associative, the $\leq$ order is reflexive, antisymmetric and transitive.

A common property in the context of semilattices is the greatest lower bound:

Definition 2.3.4 ([Aho+06])

Given a semilattice $(\mathcal{V}, \wedge)$, a greatest lower bound (or glb) of domain elements x, y is an element g such that

1. *$g \leq x$,*
2. *$g \leq y$,*
3. *If z is any element such that $z \leq x$ and $z \leq y$, then $z \leq g$.*

Common meet operators for set types are the union ($\cup$) and intersection ($\cap$) operations where the value domain $\mathcal{V}$ consists of the *power set* of the elements of the problem domain $\mathcal{U}$ (*universal set*). For example, in a reaching definitions analysis, $\mathcal{U}$ would be the set of all variable definitions.

Both $\cup$ and $\cap$ fulfill the requirement of being idempotent, commutative, and associative. The resulting semilattices can be characterized the following way:

	Set union $\cup$	**Set intersection** $\cap$
Top Element $\top$	$\emptyset$	$\mathcal{U}$ (universal set)
Bottom Element $\bot$	$\mathcal{U}$ (universal set)	$\emptyset$
Partial Order $\leq$	$\supseteq$ (set inclusion)	$\subseteq$ (set containment)

An intuitive representation of a value domain $\mathcal{V}$ is a lattice (or Hasse) diagram. The nodes represent the values $v \in \mathcal{V}$ and are connected according to the $\leq$ relation, starting with the top element. Edges denoting transitive relationships are omitted for reasons of clarity.

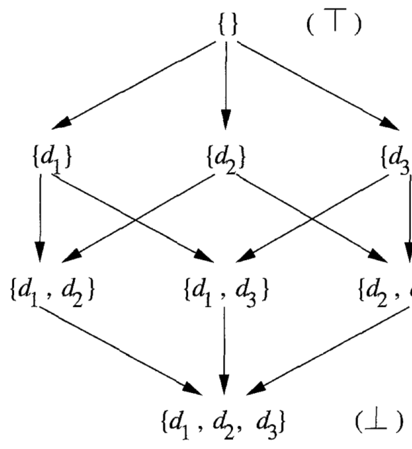


Figure 2.10.: Lattice diagram for $\wedge = \cup$ and $\mathcal{U} = \{d1, d2, d3\}$ [Aho+06].

The example shown in Figure 2.10 demonstrates this principle. In this case, the set of valid elements $\mathcal{U}$ contains $d1, d2$ and $d3$. Because the meet operator is the set union, the top element $\top$ equals the empty set (since $x \cup \{\} = x$). The diagram can be used to derive the meet (or *glb*) of two sets. This value corresponds to the highest node which is reachable on downward paths from both nodes. In the example above, the *glb* of $\{d1\}$ and $\{d2\}$ would be $\{d1, d2\}$.

The height of the semilattice plays an important role in how fast, if at all, a data-flow algorithm converges. This property is defined as the amount of edges in the longest ascending chain, i.e. one less than the amount of elements in that chain. In the presented example, the height would therefore be 3. While a finite number of values always results in a finite height, it is important to note that even semilattices with an infinite number of values may have a finite height.

2.3.2.3. Generalized Data-flow Frameworks

Based on the concepts presented above, it is now possible to specify a generalized version of *data-flow frameworks* which, according to [Aho+06], are *"algebraic structures used to encode and solve data flow problems"* involving *"a flow graph, a semilattice of values, and a set of functions from the semilattice to itself"*. It is stated that properties of these components (e.g. reducibility of the flow graph, descending chain conditions on the semilattice and monotonicity of the function space) affect the existence of an exact or approximate solution, the applicability of methods for arriving at that solution and the complexity of the method.

Definition 2.3.5 ([Aho+06])

A data-flow analysis framework $DF = (\mathcal{D}, \mathcal{F}, \mathcal{V}, \wedge)$ consists of

1. *A direction of the data flow $\mathcal{D}$, which is either* `FORWARD` *or* `BACKWARD`*.*
2. *A family $\mathcal{F}$ of transfer functions including functions suitable for the boundary conditions of the nodes* `ENTRY` *and* `EXIT`*.*
3. *A semilattice consisting of a value domain $\mathcal{V}$ and a meet operator $\wedge$.*

According to the properties of the functions, a framework can be classified as being *monotonic* or *distributive*.

Using this representation, it is possible to build a generic architecture for the specification and solving of arbitrary data-flow problems. As [KSK09] notes, *"the first benefit is that which results from any generalization. When a data flow problem is shown to be an instance of the framework, it also suggests a solution method whose properties are apparent"* eliminating the need to *"separately prove the correctness or estimate the complexity of the solution method"*. Secondly, *"generalization leads to the design of data flow analyzer generators, much in the way that lexer generators and parser generators have emerged from the study of formal languages"* by employing a *"general solution method that is parametrized with respect to the specific details of any analysis"*.

2.3.2.4. Canonical Data-flow Frameworks

A number of data-flow frameworks have been developed that can be applied to problems that are encountered in many different programming languages. To demonstrate the practical implications of the presented definitions, we will shortly describe two canonical instances, *reaching definitions* and *live variables*[9].

Table 2.1 shows the specifications of both analyses. A variable definition is a statement that assigns a value to a variable. *Reaching definitions* provide information about which definitions of a specific variable are relevant at different points in the control-flow. In [Aho+06], this property is specified as follows: *"We say a definition d reaches a point p if there is a path from the point immediately following d to p, such that d is not "killed" along that path. We kill a definition of a variable x if there is any other definition of x anywhere along the path"*. Consequently, the function gen_B denotes the definitions resulting from the assignments in block B while $kill_B$ contains all other definitions of B's variables in the program. Reaching definitions is an example of a forward analysis, since information about the definition points of variables is propagated to subsequent parts of the program.

[9] Additional examples are listed in Appendix A.2.

	Reaching Definitions	**Live variables**
Domain ($\mathcal{U}$)	Sets of definitions	Sets of variables
Direction ($\mathcal{D}$)	`FORWARD`	`BACKWARD`
Meet ($\wedge$)	$\cup$	$\cup$
Equations (IN/OUT)	$\texttt{OUT}[B] = f_B(\texttt{IN}[B])$ $\texttt{IN}[B] = \bigwedge_{P,pred(B)} \texttt{OUT}[P]$	$\texttt{IN}[B] = f_B(\texttt{OUT}[B])$ $\texttt{OUT}[B] = \bigwedge_{S,succ(B)} \texttt{IN}[S]$
Transfer Function (f)	$f(x) = (x - kill) \cup gen$	$f(x) = (x - def) \cup use$
Initialization ($\top$)	$\texttt{OUT[B]}=\emptyset$	$\texttt{IN[B]}=\emptyset$
Boundary	$\texttt{OUT[ENTRY]}=\emptyset$	$\texttt{IN[EXIT]}=\emptyset$

Table 2.1.: Two examples of common global data-flow analysis frameworks.

"In live-variable analysis we wish to know for variable x and point p whether the value of x at p could be used along some path in the flow graph starting at p. If so, we say x is live at p; otherwise, x is dead at p" [Aho+06]. In this sense, a live variable is a variable whose value is still relevant since it may potentially be read at a future point in time. The analysis of live variables is commonly used for the allocation of processor registers. The basic concepts of this analysis have been introduced by [Ken71]. If a variable is used inside a basic block (use_B), it has to be live at the block's entry point while defined variables (def_B) are dead on entry. Since we have to be concerned with variables that may reach a block on any given path, the set union is employed as meet operator. Because each program point must be "informed" about variable accesses that occur afterwards, live-variable analysis is an example of a method that relies on backward propagation of information.

2.3.3. Iterative Solving of Data-flow Equation Systems

2.3.3.1. Precision and Convergence

The total set of transfer functions forms an equation system that propagates information along the edges of the underlying control-flow graph in either the forward or the backward direction. The task of solving a DFA equation system therefore requires to compute results for the `IN` and `OUT` values of each block.

This process is complicated by the fact that, *"unlike linear arithmetic equations, the data-flow equations usually do not have a unique solution"*. Instead, the *"goal is to find the most "precise" solution that satisfies the two sets of constraints: control-flow and transfer constraints"* to determine *"a solution that encourages valid code improvements, but does not justify unsafe transformations"* [Aho+06]. From the set of valid solutions, the greatest solution with respect to the partial order $\leq$ is the most precise one.

Since loops in a program's control-flow result in a cyclic equation system, fixed-point semantics must be employed to arrive at the desired result. The best solution with respect to the problem definition is the maximum fixed-point (MFP) of the data-flow equations. A *"maximum fixed-point is a solution with the property that in any other solution, the values of IN[B] and OUT[B] are $\leq$ the corresponding values of the MFP"* [Aho+06].

For this purpose, the data-flow values are first initialized with $\top$. Then, the functions have to be repeatedly evaluated until a stable result is reached.

We will now present the round-robin and the worklist approach, two canonical methods for evaluating DFA equation systems. In general, an algorithm for solving fixed-point equation systems requires the following input:

- A control-flow graph CFG with designated `ENTRY` and `EXIT` nodes.
- A data-flow framework $(\mathcal{D}, \mathcal{F}, \mathcal{V}, \wedge)$ consisting of a direction $\mathcal{D}$, a set of transfer functions $\mathcal{F}$, a value domain $\mathcal{V}$ and a meet operator $\wedge$.
- Constant boundary values $v_{\texttt{ENTRY}}, v_{\texttt{EXIT}} \in \mathcal{V}$ for the forward and backward analysis respectively.

2.3.3.2. Round-robin Algorithm

The round-robin algorithm listed in Algorithm 1 is an example of a canonical method for solving DFA equation systems in a way that guarantees that the best solution is found. This version of the algorithm is adapted for the forward analysis case. To perform a backward analysis, it has to be modified accordingly.

Algorithm 1 Iterative round-robin algorithm

```
 1: for all (B : BLOCKS) do
 2:     if (B == ENTRY) then
 3:         OUT[B] = $v_{\texttt{ENTRY}}$;
 4:     else
 5:         OUT[B] = $\top$;
 6: repeat
 7:     stable = true;
 8:     for all (B : BLOCKS) do
 9:         if (B == ENTRY) then
10:             continue;
11:         oldvalue = OUT[B];
12:         IN[B] = $\wedge_{P \in predecessors(B)}$ OUT[$P$];
13:         OUT[B] = $f_B$(IN[$B$]);
14:         if (not OUT[B] == oldvalue) then
15:             stable = false;
16: until (stable)
```

The general idea behind this algorithm is to initialize all results with the most imprecise approximation and to repeatedly evaluate the functions to yield the final result. For this purpose, lines [1-5] assign the value $v_{\texttt{ENTRY}}$ to the initial node and $\top$ to all other nodes. In the case of a backward analysis, $v_{\texttt{EXIT}}$ would have to be assigned to the final node in the control-flow.

The algorithm then loops over the whole equation system in lines [6-16], repeatedly executing the functions, until all values have converged. Since the `ENTRY` (or `EXIT`) node has no predecessor (or successor), its value has to be computed only once. It is therefore omitted from the recomputation [9-10].

Before a new `OUT` value is computed for the current block, the old value is stored [11]. Then, the new input for the basic block is computed by applying the meet operator $\wedge$ to the output of its predecessors [12]. Afterwards, the new `OUT` value is calculated [13]. If the computed result for `OUT` is different from the stored value, the current iteration is marked as unstable, triggering the execution of a new fixed-point iteration [14-15].

[Aho+06] discusses why this algorithm is guaranteed to terminate:

> Intuitively, [the algorithm] propagates definitions as far as they will go without being killed, thus simulating all possible executions of the program. [The algorithm] will eventually halt, because for every B, OUT[B] never shrinks; once a definition is added, it stays there forever. Since the set of all definitions is finite, eventually there must be a pass of the [repeat]-loop during which nothing is added to any OUT, and the algorithm then terminates. We are safe terminating then because if the OUT's have not changed, the IN's will not change on the next pass. And, if the IN's do not change, the OUT's cannot, so on all subsequent passes there can be no changes.

With respect to the algorithm's performance, it is noted that *"[t]he number of nodes in the flow graph is an upper bound on the number of times around the [repeat]-loop"* since *"[e]ach time around the while-loop, each definition progresses by at least one node along the path in question, and it often progresses by more than one node, depending on the order in which the nodes are visited"*.

In addition, the authors mention the following properties:

1. If the algorithm converges, the result is a solution to the data-flow equations.
2. If the framework is monotone, then the solution found is the maximum fixed-point of the data-flow equations.
3. If the semilattice of the framework is monotone and of finite height, then the algorithm is guaranteed to converge.

2.3.3.3. Worklist Algorithm

The worklist algorithm represents an improvement over the round-robin method. The round-robin algorithm follows a naive approach, executing functions in an arbitrary order, not taking into account the dependencies between data-flow functions. Consequently, it has to trigger a new iteration each time a single value changes. However, the order in which the equations are evaluated can have a significant impact on the number of necessary recomputations and thus the overall performance of the algorithm. In general, it can be observed that, if the result computed for a specific data-flow value changes, only the values that directly depend on this value have to be recomputed as well. This process must then be repeated until the value of the last processed equation hasn't changed.

The worklist algorithm is a direct implementation of this approach:

Algorithm 2 Iterative worklist algorithm

```
 1: for all (B : BLOCKS) do
 2:     if (B == ENTRY) then
 3:         OUT[B] = $v_{\texttt{ENTRY}}$;
 4:     else
 5:         OUT[B] = $\top$;
 6: worklist = BLOCKS;
 7: while (not worklist == $\emptyset$) do
 8:     B = worklist.pop();
 9:     oldvalue = OUT[B];
10:     IN[B] = $\wedge_{P \in predecessors(B)}$OUT[$P$];
11:     OUT[B] = $f_B$(IN[$B$]);
12:     if (not OUT[B] == oldvalue) then
13:         worklist += successors(B);
```

Algorithm 2 describes one of multiple possible implementations of this principle. Again, this case is adapted for the forward propagation of results. Just like in the round-robin method, the equation system is first initialized [1-5]. Then, line [6] adds all nodes to the worklist since each node has to be recomputed at least once.

The main loop in lines [7-13] processes the worklist until it is empty, at which point the most precise solution, the MFP, has been found. First, a block is retrieved from the worklist [8]. After storing its old value and computing its new result, a check is performed to determine if the value has changed. If this is the case, the successors of the current block have to be recomputed as well. This is ensured in line [13] which appends the successors of B to the worklist.

2.4. Modeling

The field of modeling represents the target domain to which the analysis method that is detailed in this thesis is applied. In this chapter, we will therefore present the basic concepts and techniques of this technological space.

We start by introducing the general concepts behind the notion of modeling in Section 2.4.1. In Section 2.4.2, we present the different layers of abstraction that are of relevance in this area. Just like is the case for context-free languages, models only comprise the syntax of a language and thus another technique is required to specify its static semantics. The canonical method for this purpose is the Object Constraint Language (OCL) which will be discussed in the context of related work in Section 3.1.

2.4.1. Concepts of Modeling

Modeling languages have become a prominent instrument in the field of computer science as they enable the formalization of an application domain's concepts, their properties and the relationships between them. An abstract syntax given in the form of a metamodel validates and enforces structural constraints and fosters automated processing of the formalized information, e.g. through code generation or model transformations and the provisioning of tooling support, e.g. by building matching textual or graphical editors. In addition, the rise of modeling techniques has led to new approaches to software engineering such as the Model-driven Architecture [MDA] and Model-based Testing [AD97]. Modeling therefore represents a mechanism that can be employed in many scenarios that demand a structured representation of domain-specific data.

An important factor for the popularity of modeling techniques is that they are often perceived to provide an intuitive way for practitioners to formalize application domains. [AK03] interprets this development as the continuation of a trend that has originally been put forward by object-oriented languages: *"Instead of requiring developers to spell out every detail of a system's implementation using a programming language, it lets them model what functionality is needed and what overall architecture the system should have"*. A unified modeling methodology provides the tools necessary to (semi)automatically derive suitable editors and additional functions such as database persistence layers from a metamodel. The resulting tooling can then be handed to language users - the domain experts - who can employ this technology to solve tasks in the target domain.

Both on a conceptual level and with respect to its versatility, this concept can, for example, be compared to the widely used XML format which also organizes information according to predefined schematics, albeit with a different focus[10]: While information stored in XML is often difficult to read, models combine structured data with an intuitive (graphical) representation and are therefore capable to serve as a mediating format in the communication between humans and computers.

Arguably the most widely known modeling standard is the Unified Modeling Language (UML), *"a visual language for specifying, constructing, and documenting the artifacts of systems. It is a general-purpose modeling language that can be used with all major object and component methods, and that can be applied to all application domains (e.g., health, finance, telecom, aerospace) and implementation platforms"* [UML]. In fact, many modeling languages proposed by the OMG[11], including the UML and the Business Process Modeling Notation (BPMN), have become the de-facto standards in industry and research alike[12].

One motivation that drives the application of modeling technologies is the aspect of integration. Modeling frameworks usually define a common core, a meta metamodel, that forms the basis for the specification of modeling languages. In a sense, modeling techniques can therefore be regarded as frameworks for language develop-

[10] We discuss this in further detail in Section 4.1.1.

[11] Object Management Group (OMG) specification documents can be obtained from `http://www.omg.org/spec`

[12] An example for an alternative modeling framework can be found in KM3 [JBT06].

ment. [Bez05] compares this approach to the practices in the compiler construction domain: *"The metametamodel conforms to itself. This is very similar to the organization of programming languages. A self-representation of the EBNF notation takes some lines. This notation allows defining infinity of well-formed grammars. A given grammar, for example the grammar of the Pascal language, allows defining the infinity of syntactically correct Pascal programs"*. A notable difference between formal grammars and models is the fact that the former include the concrete syntax in the language definition while the graphical representation of model elements has to be specified separately. The similarities and differences between both fields are further explored in Section 4.1.

2.4.2. Abstraction Layers in Modeling

At the core of modeling lies the notion of different layers of abstraction which, by definition, always come in pairs: The top abstraction layer represents the metamodel while the one on the bottom conforms to an instance of this specification. A metamodel contains the declarations of concepts - commonly referred to as classes and associations - along with their properties and their relationships to other concepts. Models on the bottom layer consist of an arbitrary number of instances, each of which represents a separate entity that possesses all properties of its defining metamodel concept. In this sense, *"[a] metamodel is a model used to model modeling itself"* [MOF] and *"[t]he typical role of a metamodel is to define the semantics for how model elements in a model get instantiated"* [UMLi].

As an example, we assume that a metamodel specifies the two classes Person and Address, assigns a property name to the Person class and declares a relationship livesAt which connects both concepts. Now, on the model layer, multiple instances of these classes can be created with each instance of the Person class containing a different value in its respective name field and instances of the livesAt association indicating where each person lives.

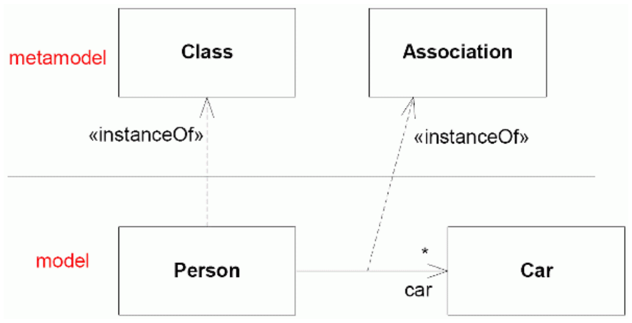


Figure 2.11.: Relationships between abstraction layers [UMLi].

It is important to realize that the distinction between the meta and the model layer always depends on the chosen viewpoint, i.e. the notion of what constitutes a concept and what represents an instance in the current context. In the example described above, the meta layer containing the definitions of Person and Address could very well be considered to be an instance of another, overlaying meta layer which specifies the properties of the Class and Association concepts themselves. From this point of view, both Person and Address would be instances of Class.

The UML infrastructure specification illustrates this principle using the example shown in Figure 2.11. In this depiction, the metamodel defines the concepts Class and Association. The model contains two instances of Class named Person and Car and one instance of Association connecting both elements. The relationships between the model and the metamodel objects are indicated by «instanceof» links that consequently span both layers of abstraction.

The involvement of more than two abstraction layers in the definition of a modeling language requires to reconsider the naming conventions with respect to the different viewpoints. Coming back to our example, we use the meta layer of Figure 2.11 to define class types for Person, Address and livesAt. One option would be to split the resulting hierarchy of three abstraction layers into two pairs. In this case, both the top and the middle layer as well as the middle and the bottom layer could be considered to be a metamodel-model pair respectively. Alternatively, we can address each layer of the three-tiered architecture from an overall viewpoint. In this case, the top level is a meta metamodel, i.e. it represents the meta layer of the metamodel, with the subjacent layers being referred to as metamodel and model respectively.

It is easy to see that, in practice, the notion of abstraction layers may lead to problems as this principle *"can be applied recursively many times so that we get a possibly infinite number of meta-layers"* [UMLi]. To avoid this complication, modeling frameworks usually specify a fixed number of layers, with the top layer supporting the recursive definition of its own concepts: *"A specific characteristic about metamodeling is the ability to define languages as being reflective, i.e., languages that can be used to define themselves"*.

The OMG architecture defines four levels of abstraction, commonly referred to as *M3*-*M0* with *M3* - the Meta Object Facility (MOF) - being the topmost layer. The UML, which resides on the metamodel layer *M2*, provides the following description:

> UML is a language specification (metamodel) from which users can define their own models. Similarly, MOF is also a language specification (metamodel) from which users can define their own models. From the perspective of MOF, however, UML is viewed as a user (i.e., the members of the OMG that have developed the language) specification that is based on MOF as a language specification. In the four-layer metamodel hierarchy, MOF is commonly referred to as a meta metamodel, even though strictly speaking it is a metamodel.

The choice of this architecture is motivated by the fact that *"many applications use proprietary models of metadata"* which presents problems as *"differences between metadata models impede data exchange across application boundaries"*. This necessitates the existence of a common basis for the development of modeling languages. A four-tiered framework realizes this requirement by extending the two-layered architecture with a meta metamodel layer *M3* that provides *"a metadata management framework, and a set of metadata services to enable the development and interoper-*

ability of model and metadata driven systems" and a bottom layer *M0* representing the real-world objects [MOF].

To provide a better understanding of the meaning of each abstraction layer, we include a shortened version of the definitions from [UMLi]:

M3 The meta metamodeling layer forms the foundation of the metamodeling hierarchy. The primary responsibility of this layer is to define the language for specifying a metamodel. The layer is often referred to as M3, and MOF is an example of a meta metamodel.

M2 A metamodel is an instance of a meta metamodel, meaning that every element of the metamodel is an instance of an element in the meta metamodel. The primary responsibility of the metamodel layer is to define a language for specifying models.

M1 A model is an instance of a metamodel. The primary responsibility of the model layer is to define languages that describe semantic domains, i.e., to allow users to model a wide variety of different problem domains, such as software, business processes, and requirements. The things that are being modeled reside outside the metamodel hierarchy.

M0 The metamodel hierarchy bottoms out at M0, which contains the runtime instances of model elements defined in a model. The snapshots that are modeled at M1 are constrained versions of the M0 runtime instances.

It is further noted, that *"[w]hen dealing with more than three meta-layers, it is usually the case that the ones above M2 gradually get smaller and more compact the higher up they are in the hierarchy. In the case of MOF, which is at M3, it consequently only shares some of the metaclasses that are defined in UML"*. This property is shared by almost all modeling frameworks, as higher levels of abstraction define very generic concepts that can be used in many different scenarios while lower abstraction layers encode detailed domain-specific information.

A simplified version of the MOF that depicts only the most important concepts is shown in Figure 2.12. The central element in this case is the Class object. A Class may contain features which are either fields that are able to hold specific values (such as the name of Person) or act as endpoints for Associations which establish links between different model elements. Additionally, Operations may be defined, representing functions that can be invoked on objects of the respective class type.

An important principle which has not yet been discussed is the notion of Generalization. A connection of this type implies an inheritance relationship between two different Classes. Similiar to the inheritance concept found in object-oriented programming languages, a specialized Class inherits all of the properties of its parent. This reduces the complexity of metamodels by avoiding a redundant specification of properties that are shared by many classes. Additionally, Generalizations carry semantic information as each specialized type implicitly possesses the type of its parent. This information can be used in scenarios such as model transformations where the class type of a model element influences the way it is processed.

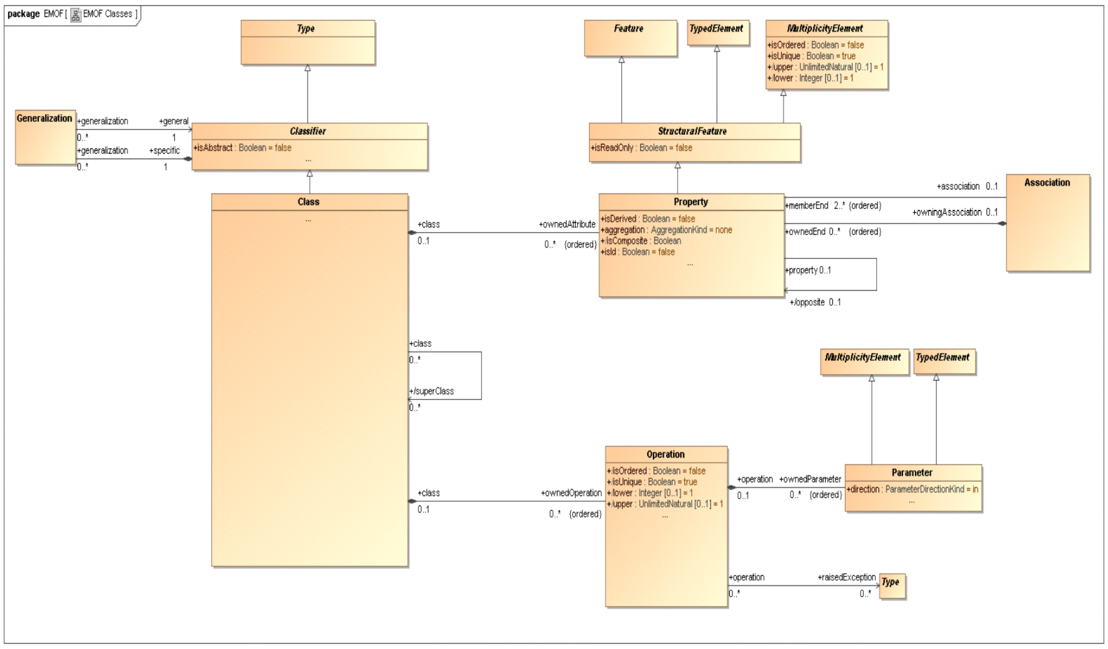


Figure 2.12.: Simplified version of the MOF [Wik08].

A concrete example spanning all four abstraction layers of the MOF framework can be seen in Figure 2.13. The right hand side of the diagram shows the artifact types that are involved in the modeling process. Based on MOF's meta metamodel on *M3*, a family of modeling languages is specified on *M2*. These comprise the Unified Modeling Language as well as user-defined languages which are referred to as domain-specific models (DSMs). Instances of these modeling languages reside on *M1* while the *M0* layer contains the data instances, i.e. the runtime (or real-world) objects in the respective target domain.

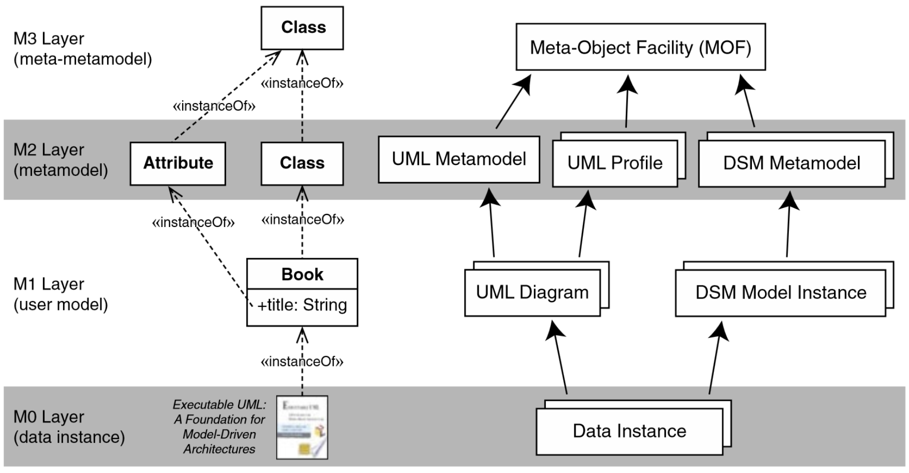


Figure 2.13.: The abstraction layers of the MOF modeling framework [ET12].

3. Related Work

In this chapter, we will investigate contemporary analysis techniques and evaluate their relationships to our own approach. Because there exist a large number of such methods, we limit our selection to techniques which are applicable in the technological space of modeling and which address issues similar to those which have been listed in Chapter 1. More specifically, we are interested in approaches which support the annotation of analysis specifications on the metamodel layer and enable a subsequent instantiation and evaluation of these statements for arbitrary language expressions.

This chapter is structured as follows: In Section 3.1, we examine the application of constraint languages for the purpose of specifying static semantics. In this context, we discuss the features of the Object Constraint Language (OCL) and the Epsilon Validation Language (EVL) with relation to the problems and challenges that have been stated in Section 1.1. The methods of attribute grammars and data-flow analysis which originate from the field of compiler construction and form the basis for the implementation of our own approach have already been covered in Section 2.2 and Section 2.3. However, since they were originally conceived, multiple proposals have been made that are aimed at combining both techniques by extending the attribute grammar formalism with support for a fixed-point evaluation of cyclic dependency paths. Furthermore, research has been carried out with the goal of establishing potential connection points between grammatical specifications and modeling artifacts. Section 3.2 provides an overview of these methods. In recent years, a wide range of existing formalisms from different technological spaces have been employed for the purpose of model analysis. Section 3.3 investigates this subject by providing an overview of relevant research work. In Section 3.4, we conclude our review of related work with a short summary and a discussion of how the presented techniques relate to the approach developed in this thesis.

3.1. Model Constraint Languages

An implicit advantage of model-based descriptions is their ability to automatically enforce certain structural constraints. This is comparable to the way a programming language's grammar specifies the valid structure of language expressions. For models, the available constructs and their structural composition are described by a metamodel (cf. Section 2.4). Unfortunately, not all constraints can be encoded in the abstract syntax of a language. This is true for context-free grammars as well as for metamodels. As a consequence, it is often the case that modeling languages require the specification of a set of well-formedness rules that represent the static semantics of the language. In the compiler construction domain, the most common method for this purpose are attribute grammars (cf. Section 2.2). In the field of modeling, the Object Constraint Language (OCL) - which formalizes structural restrictions - has a similar role.

The OMG's specification document [OCL] notes that OCL is *"a formal language used to describe expressions on UML models"* which *"typically specify invariant conditions that must hold for the system being modeled"*. It is not a programming language but *"a pure specification language; therefore, an OCL expression is guaranteed to be without side effects"*. Possible applications include the specification of invariants, pre- and postconditions, guards and constraints on operations.

Although the specification of OCL itself relies on modeling technology, constraints are written in a textual notation. Each constraint is usually evaluated in the context of a specific model element which is bound to the `self` variable in OCL's evaluation environment. For this purpose, the constraint is annotated at a class type in the metamodel and is consequently evaluated for all instances of the respective class.

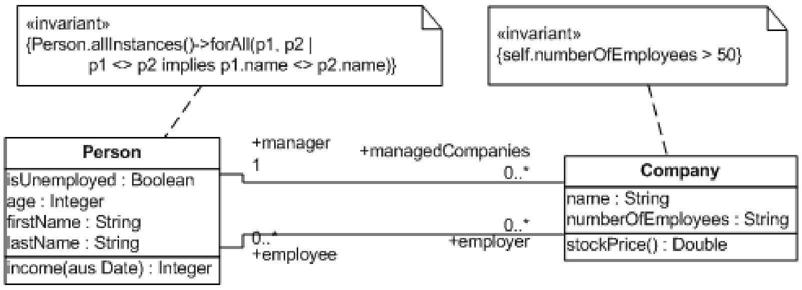


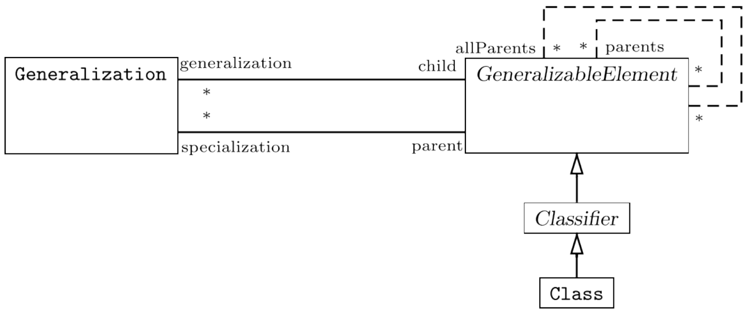
Figure 3.1.: A metamodel with two OCL constraints [OCL].

Figure 3.1 shows a simple metamodel with two OCL constraints defined in the context of the classes Person and Company. The first constraint states that the name of each person has to be unique. This is accomplished by iterating over all model elements which are of the class type Person and enforcing different values of the name property for each pair. The second constraint uses the keyword `self` to validate whether the number of employees at each instance of Company is larger than 50. During the evaluation of the constraint for objects of the Company type, the variable `self` is consequently substituted with the respective object for which the query is currently executed.

Several attempts have been made to convert UML models with annotated OCL constraints to other technical spaces, for example by translating constraints into satisfiability problems [CCR08; Soe+10]. With the availability of powerful OCL interpreters, these methods are not strictly required for constraint evaluation, however in some cases they provide additional features, e.g. snapshot generation [GBR03], to validate whether the semantics of the modeling language are preserved.

Recently, the Object Constraint Language has been extended with the ability to compute transitive closures of relationships[1], thereby addressing a fundamental shortcoming which has existed for a long time. For example, the UML infrastructure

[1] `http://www.omg.org/issues/issue13944.txt`



(a) Part of the UML metamodel, package Core

```
context GeneralizableElement
    def: parents: Set(GeneralizableElement) =
          self.generalization->collect(gen| gen.parent)

    def: allParents: Set(GeneralizableElement) = self.parents->
          union(self.parents->collect(ge| ge.allParents))
```

(b) Computing the transitive closure of parents

Figure 3.2.: Ensuring acyclic generalization hierarchies in UML [Baa03].

specification [UMLi] demands that a class must not be its own parent: *"Generalization hierarchies must be directed and acyclical. A classifier cannot be both a transitively general and transitively specific classifier of the same classifier"*. For this purpose, the following OCL constraint is given:

not self.allParents()->includes(self)

The relevant excerpt of the UML metamodel and the statements which compute the generalization hierarchy of a class are depicted in Figure 3.2(a) and Figure 3.2(b) respectively. The authors of [Baa03] state that, *"[i]nformally speaking, self.parents denotes the set of all direct supertypes and self.allParents denotes the transitive closure of direct supertypes"*. However, as [VJ00] notes, *"this operation may go into an infinite loop if there is a circularity in the parent hierarchy, in which case it is undefined"*. Making use of the closure operator, [OCL] proposes the following solution:

aClassifier.generalization()->closure(general.generalization).general()
->including(aClassifier)

It is explained that this constraint *"computes the set comprising aClassifier and all its generalizations. The closure recurses over the Generalizations to compute the transitive set of all Generalizations. The generalized classifier is collected from each of these before including the originating aClassifier in the result"*.

Alternatives to OCL exist, for example, in the form of the Epsilon Validation Language (EVL) [KPP09]. This technique has been defined on top of the Epsilon Object Language (EOL), a model management framework, which in turn *"builds on OCL"* and *"can be used both as a standalone generic model management language or as infrastructure on which task-specific languages can be built"* [KPP06].

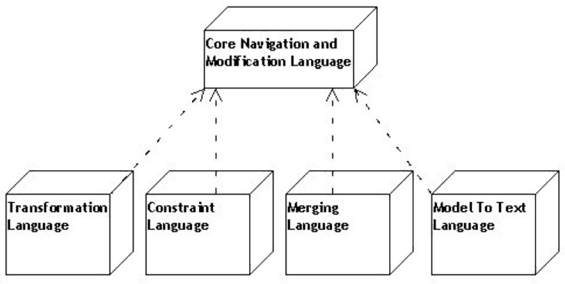


Figure 3.3.: Requirements for model management languages [KPP06].

Model management tasks include the transformation of models as well as consistency checking. To provide a unified foundation for these scenarios (cf. Figure 3.3) the authors of [KPP06] have devised a number of characteristics which must be supported by the language core such as support for navigation, modification and cross-model operations. This endeavor is motivated by several shortcomings of OCL, for example the lack of support for statement sequencing, inter-model operations and unnecessary diversity in navigation expressions. [KPP09] lists additional problems such as the lack of support for user feedback and the fact that *"[e]ach OCL invariant is a self-contained unit that does not depend on other invariants"*.

A similar approach is taken by [JB05] which implements a language for consistency checks on top of the ATLAS modeling framework [Jou+06].

3.2. Circular Reference attributed Grammars

In Section 2.2 we described the method of attribute grammars, a technique which is commonly employed in the field of compiler construction for the validation of the static semantics of programming languages. For this purpose, semantic attributes are annotated at grammatical symbols and can therefore be subsequently instantiated for parsed language expressions. Based on the type of the attribute - which is either classified as inherited or synthesized - this results in a top-down or bottom-up information flow along the edges of the respective syntax tree. Since these edges represent the application of grammatical derivation rules, information is routed between parent and child expressions, thereby evaluating each subexpression in its full syntactical context. The relevance of the AG approach with respect to the objectives of this thesis can be characterized as follows: Attribute-based analysis specifications enable the enrichment of language definitions with declarative analyses which rely on the syntactical structure of language expressions to propagate flow-sensitive information.

In their traditional form, as proposed by [Knu68], attribute grammars however do not provide evaluation semantics for cyclic equation systems resulting from circular dependencies. Consequently, as stated, for example, in [Sag+89], *"until recently, circular AGs (CAGs) were considered ill formed and meaningless"*. This limitation can be explained by the origins of the attribute grammar formalism. The validation of static semantics usually requires the computation of a unique solution which determines whether the respective expression is either correct or invalid. In contrast to this requirement, cyclic attribute grammars may yield an arbitrary number of solutions. In general, this behavior is more closely associated with approaches for program optimization - such as data-flow analysis - which are specifically designed to compute (conservative) approximations of runtime behavior. It can also be argued that, since the information flow in attributed syntax trees follows the derivation hierarchy of language expressions rather than the control-flow of the program, it is more difficult to specify meaningful analyses which require an iterative fixed-point computation. Finally, many variants of the attribute grammar formalism put an emphasis on a performant execution. A wide variety of optimizations exist that are aimed at facilitating this property ranging from special classes of attribute grammars which can be computed during the syntax analysis phase of the compile process to methods which enable the static creation of optimal attribute evaluators for arbitrary language expressions. Iterative computations, which are required for deriving the fixed-point of a cyclic equation system, would complicate the application of these methods since it is more difficult to statically determine the order in which the semantic attributes have to be evaluated and multiple passes may be necessary until the final result is available.

Nevertheless, multiple proposals have been made to extend the attribute grammars formalism with the required constructs, semantics and evaluation strategies to specify and compute results for circular AGs. This has been explored, for example, by [Rod86] where it is argued that the definition of circular dependencies may indeed result in valid and meaningful specifications: While *"in the traditional formulation of attribute grammars (AGs) circularities are not allowed [...] elsewhere in mathematics and computing, though, circular (or recursive) definitions are commonplace, and even essential"*. It is argued that support for circular dependencies (using fixed-point semantics for the evaluation of circular dependencies in attributed syntax trees) can actually be very useful under certain circumstances. The author emphasizes the importance of the AG mechanism by asserting that *"one of the major strengths of AGs is that they are non-precedural specifications"* and concludes that this also applies to the case of circular definitions: Although they do not necessarily possess a unique solution, a *"recursive definition of attributes is even more of a specification"* as *"it describes conditions that the attributes must satisfy"*. This claim is motivated by the assertion that traditional problems such as *live variable analysis* can be defined and executed using circular attribute grammars. For this purpose, the author introduces the classes of recursive and finitely recursive AGs along with an algorithm which is able to derive a static evaluator for these cases. With respect to the evaluation process, it is stated that *"the author of a recursive AG should not have to supply algorithms for constructing the least fixed-point"* but rather only *"have to supply enough information to determine that such fixed-points do exist and can be computed"*. With this assertion, the author emphasizes that circular attribute grammars are able to provide a methodology which addresses a greater range of problems while assigning the responsibility for ensuring that the computed results will represent a meaningful solution to the developer.

[Sag+89] emphasizes the importance of providing support for the specification and evaluation of circular attribute grammars. In this paper, the authors claim that common data-flow analysis problems can in fact be encoded using the AG formalism and that an evaluation mechanism which supports fixed-point computations would therefore enable an attribute-based implementation of these algorithms: *"[I]f the semantic equations employ monotonic operators [. . .], they define a unique greatest (least) fixed point which may be interpreted as the meaning of the equations"* and that *"this kind of circularity arises naturally in many data flow analysis (DFA) problems"*. Since, *"for structured languages, various DFA problems [. . .] have been specified using CAGs"*, their method focuses on automating *"the translation for most of the data flow analysis problems considered since most of them could be easily specified using uniform modification CAGs"*. In contrast, [Ros90] explores the aspect of abstract interpretation using circular attribute grammars and in this context mentions that *"the use of attribute grammars for the specification of abstract interpretation can be seen as an alternative notation to denotational semantics"*.

Another extension of the AG formalism can be found in reference attributed grammars (RAG) [Hed00]. While traditionally, values are only exchanged between neighboring elements in the parsed language expression, the reference mechanism *"permits attributes to be explicit references denoting nodes arbitrarily far away in the syntax tree"*. Consequently, an information flow between arbitrary nodes can be established as a *"reference value denoting a node in the syntax tree may be dereferenced to access the attributes of that node"*. As a result, *"information can be propagated directly from the referred node to the referring node, without having to involve any of the other nodes in the syntax tree"*. In this case, the dependency graph between attribute instances superimposes the structure of the syntax tree by allowing the specification of additional cross-edges. As noted in [MH03], the method of RAGs can be further enhanced by combining the declaration of reference attributes with a demand-driven evaluation semantics for circular dependencies. This approach allows not only to compute fixed-point results for circular attribute grammars but also enables the specification of arbitrary data-flow paths through the use of attributes which are able to address values located at remote nodes. The resulting formalism - which is referred to as circular reference attributed grammars (CRAG) - allows *"the recursive definitions to be specified directly in the grammar, and the fixed point to be computed by an automatically generated evaluator"*.

The authors of [Bur+11] note that CRAGs provide an opportunity to *"alleviate the lack of support for formal semantics specification in metamodelling"* by applying this technique for the purpose of model analysis. The resulting approach - called *"semantics-integrated metamodelling"* - therefore implements a methodology similar to the flow-based analysis technique detailed in this thesis. More specifically, the authors identified similarities and differences between the technological spaces of modelware and grammarware with the goal of transferring compiler construction techniques to the modeling domain. By employing the CRAG formalism as underlying specification method, remote and circular attributes can be used to define and evaluate cyclic information flows.

For this purpose, the paper investigates how the syntax and semantics of metamodeling languages can be mapped to the CRAG formalism. In the first step, an

Syntax in Ecore	Syntax in JastAdd
EClass	Node Type
EReference [containment]	Non-Terminal Child
EAttribute [non-derived]	Terminal Child

E_{syn}

Semantics Interface in Ecore	Semantics in JastAdd
EAttribute [derived]	synthesized attribute, inherited attribute
EAttribute [derived, multiple]	collection attribute
EReference [non-containment]	collection attribute, reference attribute
EOperation	synthesized attribute, inherited attribute

E_{sem}

Figure 3.4.: Integrating Ecore and JastAdd [Bur+11].

existing MOF-based metamodel has to be translated to a context-free grammar. This can be accomplished by mapping the classes of the metamodel, their attributes and their containment hierarchy to grammatical productions. Since cross-references, i.e. links which are not containment relationships, cannot be represented by a CFG, the authors classify these elements as semantic properties of the target language and generate appropriate semantic attributes which can be annotated at the respective symbols. Figure 3.4 shows the derived relationships between metamodel and grammar concepts. The elements are divided into two groups: While concepts in the set E_{syn} are deemed to be syntactical properties as they can be directly translated into corresponding grammatical representations, the semantic properties in E_{sem} contain types which have to be implemented as semantic attributes. The resulting specification represents the target metamodel in the form of a semantically enhanced grammar. Based on this grammar, models can be represented as syntax trees with a superimposed reference graph that encodes the elements of E_{sem}. It is now possible to extend the derived CRAG with custom analysis specifications in the form of semantic attributes.

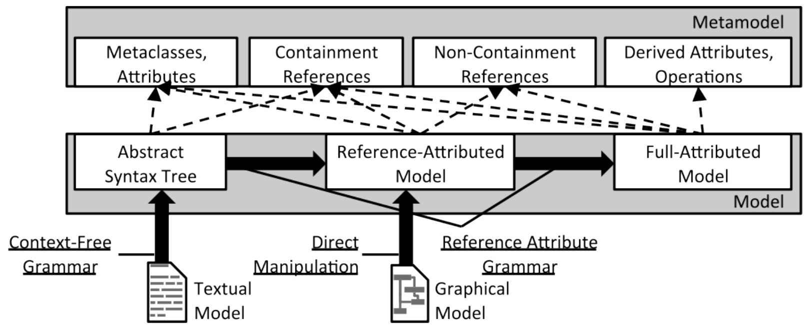


Figure 3.5.: Transformations in metamodeling [Bur+11].

Figure 3.5 illustrates the process of constructing the respective CC artifacts based on their modeling counterparts. In the first step, an abstract syntax tree is derived from the classes, attributes and containment references. Next, the non-containment references are translated into reference attributes resulting in a *reference-attributed model*. In the final step, the grammar specification is enriched with derived attributes and class operations as defined in the metamodel.

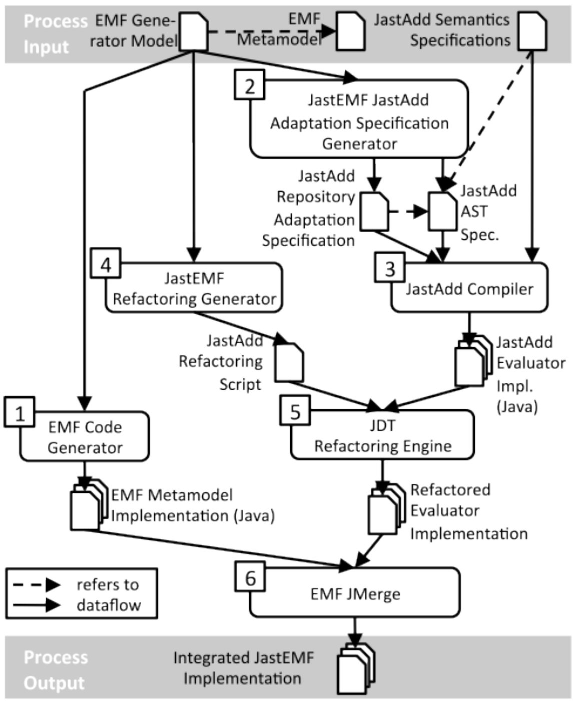

Process Input
EMF Generator Model
EMF Metamodel
JastAdd Semantics Specifications
2 JastEMF JastAdd Adaptation Specification Generator
JastAdd Repository Adaptation Specification
JastAdd AST Spec.
4 JastEMF Refactoring Generator
3 JastAdd Compiler
JastAdd Refactoring Script
JastAdd Evaluator Impl. (Java)
1 EMF Code Generator
5 JDT Refactoring Engine
EMF Metamodel Implementation (Java)
Refactored Evaluator Implementation
refers to
dataflow
6 EMF JMerge
Process Output
Integrated JastEMF Implementation

Figure 3.6.: JastEMF's Generation Process [Bur+11].

An implementation of this approach is available in the form of the JastEMF framework. This tool uses EMF (cf. Section 7.1.2) as a basis for the implementation of model-related functionality and the JastAdd metacompiler system [EH07] - which provides an implementation of the CRAG formalism - as execution platform for the derived analyses. The overall process is depicted in Figure 3.6. Based on an EMF metamodel, an implementation is generated ① along with an JastAdd AST specification ②. This artifact is combined with additional semantic specifications from which the JastAdd compiler ③ creates a corresponding attribute evaluator. The remaining steps integrate the resulting artifacts on a technical level.

3.3. Other Related Work

It should be mentioned, that attribute grammars and data-flow analysis are not the only techniques which can be used to analyze programs. In the following, we will provide a short summary of several common analysis paradigms as listed in [KSK09]:

Inference Systems

These systems infer properties through a repeated application of inference rules. For this purpose, a set of axioms and inductive rules is provided while the task of choosing the appropriate rules is up to the user.

Constraint Resolution Systems

Here, constraints are written as inequalities and the evaluation requires the identification of a solution which satifies these statements.

Model Checking

If a program is represented as a formal model, the properties can be expressed as boolean formulae which are then computed by a model checker.

Abstract Interpretation

Based on mappings between concrete and abstract semantics, soundness of the abstraction functions is validated using abstract interpretation theory.

While none of these techniques was originally conceived with the goal of an application in the modeling domain, various research work has emerged which makes use of these methods for the purpose of model analysis. One example is the Alloy framework [Jac02] which has been employed in multiple use cases. According to [MGB04], *"Alloy is a formal modeling language based on first-order logic, allowing specification of - primarily structural - properties in a declarative fashion"*. It is further stated that *"models in Alloy are described at a high level of abstraction"* and that *"one can apply object modeling in a similar fashion to UML class diagrams"* while *"allowing automatic analysis"*. An Alloy model consists primarily of signatures corresponding to complex types and logical formulae which describe restrictions over instances of these types. As mentioned in [JSS00], *"Alloy is not a decidable language"* and it is therefore not possible to *"provide a sound and complete analysis"*. Instead, the basic premise of this approach is to *"search within a finite scope chosen by the user that bounds the number of elements in each primitive type"*. For this reason, Alloy is usually classified as a *model finder*. [VJ00] discusses the relationships with the OCL, arguing, for example, that Alloy provides a simpler notation. For the computation of the transitive closure of class hierarchies, the following definition is given:

```
all e: GeneralizableElement | e.allParents = e.+parent
```

with an additional constraint that checks for circularity without going into an infinite loop:

```
all e: GeneralizableElement | e !in e.allParents
```

According to [VJ00], additional advantages include self-contained expressions which do not rely on external (meta)models and purely constraint-based specifications which do not allow for operational statements.

Multiple papers have been published which deal with the application of Alloy for the analysis of UML models, amongst others [Ana+07; Ana+10; SAB10; MGB04]. For this purpose, [Ana+07] describes how a subset of UML/OCL can be mapped to Alloy's specification language to identify inconsistencies in the definition of the target models. The authors list a number of challenges which complicate the transformation process. For example, while Alloy supports generalization, it does not enable redefinition at subclasses. It is also stated that, due to the lack of a concept such as `self` in OCL, it is *"difficult to reference the instance of the signature on which the Alloy predicate is applied"*. [Ana+10] further notes that *"model transformation from UML to Alloy has proved to be very challenging"* as *"UML and Alloy* *have fundamental differences, which are deeply rooted in their underlying design decisions"*.

The authors of [SAB10] describe how analysis results, i.e. model instances which represent counter-examples to the stated constraints, can be related back to the original modeling language. The paper presents a method for automatically generating transformation rules that are able to convert these instances into UML Object Diagrams. It is mentioned that *"[defining] a transformation from Alloy to UML to carry the outcome of analysis conducted by Alloy to an object diagram has proved to be challenging"* since the translation from UML to Alloy involves the M2 layer while the results correspond to M1 models. Because the translation is not an 1-to-1 mapping, *"[i]nstances in Alloy are not naturally instances of the originating UML model - some information is 'lost in transformation"'*. As a solution to this problem, execution traces of the UML2Alloy[2] transformation are used to dynamically generate suitable translation instructions. Case studies for the application of Alloy in the modeling domain include the identification of inconsistencies in models through automatic snapshot generation [MGB04] and the representation and validation of model transformations [ABK07].

A similar approach is taken by the USE tool [GBR07]. [Gog07] states that *"like Alloy [USE] is more a model finder than a model checker, i.e., it helps developers in finding models with desired properties"*. According to the authors, their tool can be applied to validate the consistency of model and constraint specifications by constructing respective instances. Additional scenarios include checking for the independence of invariants and testing whether undesired properties may assert themselves in instantiations. [GBR03] describes how this method enables the generation of complex system snapshots based on OCL descriptions.

In addition to generic techniques which support the specification of different kinds of model analyses, there are also many publications which employ methods that are tailored to solving a very specific problem. For example, the approach detailed in [GBL05] converts UML Sequence Diagrams to Concurrent Control Flow Graphs (CCFG). A subsequent flow analysis on this model-based representation is used to derive Concurrent Control Flow Paths (CCFP). It is suggested that these results can be used in sequence-based testing approaches. A related application scenario for data-flow analysis is addressed by [BLL10]. The authors describe how control-flow information can be derived from UML State Machines to drive the selection of cost-effective test cases. The usefulness of flow-sensitive information in the area of software testing is also explored in [RW82]. In [WIM08], the authors employ the action semantics of Executable UML (xUML) models for the specification of software systems. Subsequent usage of data-flow analysis enables an evaluation of the abstract semantics of the executable specifications with the goal of *"[finding] def-use associations among the actions written in an action language"*.

[2] http://www.cs.bham.ac.uk/~bxb/UML2Alloy/

3.4. Summary and Discussion of Related Work

In this chapter, we introduced different techniques which support the specification and execution of model analyses. We will now conclude our review of related work with a discussion on how the presented methods relate to our own approach on a conceptual as well as on a technical level.

	OCL	EVL	CRAG/JastEMF	Alloy	DFA for Models
Specification					
Formal specification	✓	×	✓	✓	✓
Model-based specification	✓	✓	×	×	✓
Inter-language links	✓	✓	×	×	✓
Reuse of definitions	○	○	×	×	✓
Redefinition at subclasses	○	○	○	×	✓
Capabilities					
Transitive closure	✓	✓	✓	✓	✓
Information propagation	×	×	✓	×	✓
Fixed-point semantics	×	×	✓	×	✓
User feedback	×	✓	○	×	○
Snapshot generation	×	×	×	✓	×
Syntax					
Textual syntax	✓	✓	✓	✓	✓
Graphical syntax	○	×	×	×	×
Other					
Tool support	✓	✓	✓	✓	✓

Table 3.1.: Comparison of different static analysis methods.

In the following, we will compare the described approaches, focussing on several characteristic properties to determine how these techniques relate to the method proposed in this thesis. For this purpose, we perform a side-by-side comparison on a set of aspects which mirror the goals listed in Section 1.2. It should be noted that this evaluation is not an exhaustive review of the full feature set of the included approaches but rather focuses on aspects which are relevant to the realization of the intended application scenarios. The results of this assessment are shown in Table 3.1. Support for properties is rated in three stages: Checkmarks and crosses denote whether the respective requirements are met while a circle indicates partial support.

The characteristic properties are grouped into four categories. The first category addresses issues which relate to the underlying principles of the respective method. This includes the formal basis of the approach itself as well as the methodology for analysis specification.

Formal/Model-based specification

The first two items state whether the syntactic concepts and the evaluation semantics are supported by a formal and/or a model-based specification. The former property requires that the technique has a sound mathematical foundation while the latter indicates whether model-based descriptions are available for the relevant analysis concepts. An example would be a metamodel which encodes an analysis DSL. Formal definitions are a prerequisite for a clear and unambiguous understanding of the methodology while model-based specifications facilitate the integration of analysis functions with standards and tools of the MDE domain on a conceptual as well as on a technical level.

Inter-language links

This property denotes whether analysis concepts can be directly associated with modeling artifacts. If this is not the case, this results in a gap between the modeling and the analysis space. Bridging this gap usually requires a variety of (interconnected) transformation steps. Since these must be specifically adapted to each target modeling language, this can substantially increase the complexity of the respective approach. Differences in the semantics of the modeling and the analysis space may further complicate this process. Not only may this lead to a loss of semantic information, but diverging notions on how certain concepts are to be interpreted in each space prevent a consistent and intuitive application of the respective methodology.

Reuse of definitions

Reuse is an important aspect as it affects development and maintenance efforts. This is especially true if a method is applied more than once under similar circumstances. This principle can refer to the reuse of complete analyses, adaptable templates or smaller fragments of complex specifications. Advanced support can be provided through a sophisticated library concept which enables the organization of existing analyses in functional packages.

Redefinition at subclasses

The notion of generalization is very common in the modeling domain. Typically, it is assumed that properties and features of a class are inherited by all subtypes. Modeling frameworks such as MOF additionally support the redefinition of inherited concepts to enable specialized elements to implement a different behavior than their parents. This can be compared to the overwriting of inherited methods in object-oriented languages. This feature can be very useful for the specification of model analyses since the evaluation semantics for subclasses may differ from the interpretation of their parents.

The next category describes capabilities which affect the expressiveness of the analysis methodologies.

Transitive closure

Computing the transitive closure over a relationship is a feature which is useful in many application scenarios. In general, this function can be regarded as a special case of information propagation (see below) which is often supported through dedicated operators. In practice, deriving meaningful closures may require more complex specifications which include multiple relationship types or take conditional statements into account.

Information propagation

The propagation of information inside the target models enables analyses to abstract from the layout of the underlying modeling language. This is important if an analysis cannot presume a specific structure on the metamodel or model layer and therefore cannot employ fixed navigation expressions to address distant elements. A simple example would be the exchange of information between nodes in a control-flow graph which may be separated by an arbitrary number of steps. In contrast to the computation of transitive closures, this principle provides the means to implement varying propagation semantics for different metatypes and can therefore be very useful in the modeling domain.

Fixed-point semantics

Support for fixed-point evaluation semantics is an important prerequisite for the static approximation of dynamic behavior based on the structural composition of language expressions. This feature usually coincides with the use of information propagation as the presence of cyclic propagation paths necessitates a dedicated handling to arrive at a meaningful solution. For obvious reasons, it is advantageous if the respective solver can detect cyclic dependencies and automatically apply fixed-point semantics if required.

User feedback

Once an analysis has been successfully executed, the final verdicts must be reported to the user in a suitable fashion. Since the raw results are often difficult to interpret, they must be refined in a postprocessing step. Alternatively, analysis methodologies may provide inbuilt functions for generating meaningful outputs. This requires that the results indicate both the nature and the location of violated constraints. Advanced feedback mechanisms may additionally enable the specification of automatic fixes for identified problems.

Snapshot generation

Approaches which support the generation of instance snapshots can be used to validate the correctness of the modeling language itself or to create prototypic models for application scenarios such as model-based testing. For this purpose, metamodels are typically enriched with specifications of (un)desired properties. A model finding algorithm then tries to generate instance candidates which fulfill or contradict these restrictions.

The final two categories state whether analyses can be specified in a textual and/or graphical notation and whether an approach is supported by tooling. In the following paragraphs, we will discuss the classification of the different methods in more detail.

OCL/EVL

Because of their origin as constraint languages for the MDE domain, it is not surprising that both the OCL and the EVL are firmly rooted in modeling principles. Since the abstract syntax is defined in the form of a metamodel, the constraints conform to model-based abstract syntax trees which, by design, are directly connected to the target model artifacts. This integrative approach also prevents semantical gaps between the analysis space and the modeling domain. Nevertheless, redefinition of constraints is only partially supported as a *"subclass may strengthen the invariant but cannot weaken it"* [WK03]. While there exist several formalizations of OCL's semantics (e.g. [CK04a]) the version included in the official specification document [OCL] is outdated as it is based on an older version of the standard. While the OCL does not provide native support for the management of constraints, it is possible to realize a rudimentary library concept by separating modeling artifacts and analysis specifications. Since both OCL and EVL were conceived as traditional constraint languages, they do not support information propagation or the computation of fixed-point results. Nevertheless, they include dedicated operators for computing the transitive closure of relationships as this has been discovered to be a very useful feature. In contrast to OCL, EVL provides sophisticated methods for encoding feedback messages and even fixes for erroneous models in the analysis specification.

CRAG/JastEMF

The JastEMF tooling is based on the CRAG formalism which extends attribute grammars with support for circular dependencies and fixed-point semantics. It therefore builds on a well-founded theoretical framework. However, rather than enhancing this method with native support for the MDE domain, the approach relies on multiple transformation and mapping steps to adapt modeling artifacts to the requirements of the analysis space. Because of the necessary transformation steps and missing inter-model links, this can present a challenge to tooling providers. The authors of [Bur+11] do not mention whether redefinition of attributes at subclasses is supported but it can be assumed that this feature can be realized by associating subtypes with different semantic attributes. The underlying CRAG formalism natively supports the principles of information propagation and fixed-point semantics and can compute the transitive closure of a relationship. However, the conceptual differences between formal languages and models complicate the analysis specification process as the user is exposed to technical details such as the distinction between inherited, synthesized and circular attributes. Since the specifications are realized as aspect-oriented attributions, it is generally possible to build functional libraries to facilitate the reuse of existing analyses. Although the results of the validation are implicitly visualized as properties of the respective model elements, JastEMF does not seem to include a dedicated methodology for the generation of meaningful feedback messages.

Alloy

The Alloy framework relies heavily on mathematical formalisms such as set theory and first-order relational logic. Just like in the JastEMF approach, the conceptual gap between the analysis space and the target domain must be bridged using a variety of transformation steps. As noted, for example, in [Ana+07] and [Ana+10], this process is further complicated by differences in the underlying semantic concepts. [Ana+10] states that, amongst other divergences, *"Alloy does not directly support the notion of redefinition. More specifically signatures that belong to the same hierarchy may not define fields with the same name"*. It can be argued that, although the principles of information propagation and fixed-point evaluation are not supported, these features would only play a minor role in the intended application scenarios as this analysis framework primarily functions as a model finder. In a general sense, the textual notations of analysis specifications offer the possibility to organize functions in libraries. However, the fact that this representation integrates modeling and analysis artifacts may complicate this process in practical applications. Due to the absence of a native support for model-based specifications, it is difficult to correctly represent the generated model snapshots as valid instances of the original modeling language [SAB10]. While the listed publications propose partial mappings for the UML, this process has to be repeated for every target DSL.

Conclusions

From the previous discussion, we can conclude that neither of the existing methods is able to fulfill all of the initially requested requirements and design goals. While each approach has its specific application scenarios and (dis)advantages, none of them implements an integrated, model-based analysis methodology which is well-suited for the intended purposes.

Both the OCL and the EVL focus on the (side-effect free) validation of structural constraints. They are however not well-suited for the derivation and approximation of context-sensitive information - a limitation removed by the combination of the information propagation principle and the fixed-point evaluation semantics of the data-flow method. The CRAG formalism and its adaption to the modeling domain, the JastEMF toolset, heavily rely on formalisms traditionally employed in the field of compiler construction. As a result, the processes of analysis specification and execution are driven by the requirements of the underlying attribute evaluator rather than the properties of the modeling domain. In conclusion, both JastEMF and the data-flow analysis inspired methodology are based on similar motivations and usage scenarios but approach the problem from different starting points. While JastEMF transforms modeling artifacts for an application of traditional compiler construction techniques, we adapt the techniques themselves to provide native support for the characteristics of the modeling domain. Alloy differs from the other approaches in the sense that it functions as a model finder which searches for M1 candidates that violate stated restrictions. Nevertheless, this methodology supports the implementation of a range of validation scenarios. However, due to the semantic gap between the analysis and the modeling space and the requirement to develop custom mappings for each DSL, the practical applicability of this technique for the purpose of model analysis is limited. The described methods also include approaches which implement a flow analysis inspired evaluation of modeled artifacts. These publications address specific problems and do neither provide nor rely on a generic analysis methodology that supports the specification of arbitrary analyses. In fact, it would be possible to realize analyses such as [GBL05] using our generic approach. This is, for example, demonstrated in Section 10.1.4 which describes DFA-based reimplementations of the analyses developed by [Got+09; Gar08].

Part II.

Data-flow based Model Analysis

4. Adapting Flow Analysis to the Modeling Domain

While the description and validation of the static semantics of formal languages is a well-researched field, the same cannot be said about the domain of modeling. A testament to this shortcoming can be found in many OMG specifications. Although several of the techniques proposed by the OMG have become the de facto standard in the modeling domain, they still rely to a great deal on informal descriptions of semantic constraints.

UML's *superstructure specification* [UMLs] tries to alleviate this problem by formalizing semantic constraints using the Object Constraint Language. For example, the restriction *"A constraint cannot be applied to itself"* is given as `not constrainedElement->includes(self)`. However, it is also often stated that a specific constraint *"cannot be expressed in OCL"* in which case only an informal textual description is given. The lack of a powerful method for encoding semantic properties not only results in obvious problems when dealing with the task of validating UML models, it also affects designers of domain-specific modeling languages who attempt to formalize the semantics for the purpose of validating language expressions or evaluating their static properties.

The proposed solution to this shortcoming is a model-centric analysis methodology based on the traditional approaches of data-flow analysis and attribute grammars which are heavily used in the discipline of compiler construction. Models and formal languages (cf. Section 2.1.2 and Section 2.4 respectively) both support the specification of an abstract syntax, albeit with differences in their conceptual approaches and application scenarios. Therefore, transferring these methods from one domain to the other requires a careful consideration of the involved principles and formalisms. For this purpose, we will juxtapose both areas, highlighting differences and similarities and discuss how the findings relate to the objective of this thesis.

The task of the conception and implementation of an approach to static model analysis has to begin with a careful evaluation of the explicit and inherent properties of the domains of modeling and formal languages. This is the subject of Section 4.1. Based on the results of this comparison, Section 4.2 details the requirements for the adaption of the existing and well-tested semantic validation techniques. From these design goals and the identified relationships between the areas of modeling and compiler construction, we then derive a list of specific challenges and objectives for the transfer of the DFA method to the modeling domain. This discussion serves as a basis for the formal and technical specifications presented in the following chapters.

4.1. Relations between Formal Languages and Models

A detailed description of the adaption of the compiler construction techniques of data-flow analysis and attribute grammars to the modeling domain will be given in Chapter 5 and Chapter 6. To provide a sound basis for these definitions, it is necessary to discuss the conceptual alignment of these two fields in advance. This comparison has to be carried out carefully since *"[b]esides the fact that a metamodel is usually graph-based and a grammar tree-based, there are several differences in application between these notions"* and because *"[s]ome metamodels may be similar to grammars, but generally they are more versatile"* [Bez05]. It is therefore vital to identify and categorize the properties which affect the alignment of both techniques and to discuss the respective similarities and differences.

For this purpose, this section examines the relationships between both areas:

Formal languages have been introduced in Section 2.1. More specifically, the description focused on the subclass of context-free languages which play a key role in the context of compiler construction. As will become evident, this class of formal languages has strong conceptual ties to the modeling area.

Modeling techniques have been presented in Section 2.4. They form the basis of model-driven engineering and represent the technological space to which the methods for static program analysis should be applied.

Section 4.1.1 introduces the notion of technological spaces to motivate the alignment of both domains. In Section 4.1.2, we examine both fields in the context of their respective application domains, i.e. compiler construction and software/domain engineering. This includes a discussion of the recent trend towards language engineering vs. the traditional use of predefined languages and how this development facilitates the development of techniques that transcend both areas.

Expressions that have been constructed according to a context-free grammar follow the structural restrictions defined by grammatical rules, just as models comply with the abstract syntax given by a metamodel. The involved abstraction layers as well as their alignment and a subsequent comparison are the subject of Section 4.1.3.

Since languages are usually designed for use by humans, their textual and/or graphical representation plays an important role in the specification process. The same applies of course to any analysis language which operates on a target language's constructs[1] and is therefore not only connected to the language on a definition but also on a representational (or visual) layer. In Section 4.1.4 we identify the conventions that exist in this context and explore the consequences relevant to the advancement of our approach.

In addition to the syntactical definitions of a grammar or a metamodel, the static and dynamic semantics of a language are equally important aspects that have to be considered when specifying a language. This is the subject of Section 4.1.5.

[1] In this case, the constructs of the target language are the constituents of a either a metamodel or a grammar.

4.1.1. Technological spaces

The authors of [KBA02] propose a classification along the line of so-called technological spaces (TSs)[2], stating that a technological space is a *"working context with a set of associated concepts, body of knowledge, tools required skills, and possibilities"* which *"is often associated to a given user community with shared know-how, educational support, common literature and even workshop and conference meetings"*.

In this thesis we will use the refined definition given in the follow-up paper [B+05]:

Definition 4.1.1

A technical space is a model management framework accompanied by a set of tools that operate on the models definable within the framework.

The relevance of these considerations stems from the fact that *"a problem inside one particular technology, [. . .] corresponds to similar problems within other technologies"* although *"the way to solve this problem is not always the same"*. Examples given include programming languages, database systems (DBMS), frameworks for markup languages (XML, SGML) and knowledge representation techniques such as ontologies as well as modeling frameworks and languages.

Relationships between Technological Spaces

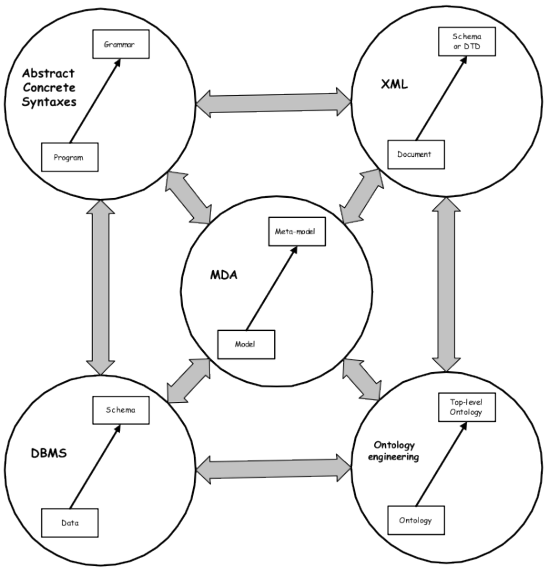
The authors of [B+05] motivate the study of technological spaces with following two observations:

- [D]espite the different purposes of technologies they expose some common characteristics. Studying these commonalities and capturing them in a general concept can help the understanding of a new technical space and can provide a framework for comparison among technical spaces.
- [O]ften various technical spaces offer complimentary features. A problem at hand may be solved in an easier way by using collaboration between different techniques. Therefore, software engineers should be aware of the strong and weak points of every technology and should be able to make conscious choices among them.

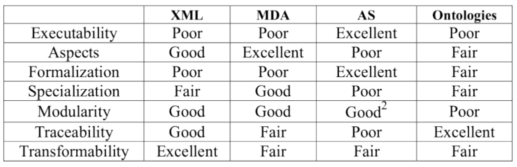
These are not isolated areas but rather can be connected through bridges as visualized in Figure 4.1(a). It is essential to note that these bridges are facilitated by a common property of the technological spaces: *"[M]any technical spaces organize its artifacts across a layered architecture based on the notions of model, metamodel, and metametamodel (so called three-level conjecture). This observation is used to give a semi-formal definition of the concept Technical Space"*.

Effectively, this means that each space utilizes multiple abstraction layers - often identified as meta and instance layers - to provide formalized language frameworks

[2]The terms technical space and technological space will be used interchangeably.



(a) Bridging technological spaces



	XML	MDA	AS	Ontologies
Executability	Poor	Poor	Excellent	Poor
Aspects	Good	Excellent	Poor	Fair
Formalization	Poor	Poor	Excellent	Fair
Specialization	Fair	Good	Poor	Fair
Modularity	Good	Good	Good^2	Poor
Traceability	Good	Fair	Poor	Excellent
Transformability	Excellent	Fair	Fair	Fair

(b) Properties of technological spaces

Figure 4.1.: Examples of technological spaces, their relations and properties [KBA02]

comprising standards, methodologies and tools. These frameworks do not necessarily correspond to the notion of formal languages as it is understood in theoretical computer science but rather are methods that enable the definition of constructs in a either a specific domain or using a specific technology. By using formalized descriptions, the derived language expressions can be subjected to different kinds of automated processing for purposes such as static validation or transformation. This principle is demonstrated in Figure 4.1(b) which compares properties of several technologies to determine how well the respective framework performs in different areas. The multiple levels of abstraction relate each technological space to each other on a conceptual level and thus form the basis for the implementation of technology bridges[3]. The alignment of abstraction layers is further explored in Section 4.1.3.

The relevance of the MDA (or model-driven engineering in general) in particular is stressed by [SB05]: The authors not only categorize MDA as a technological space but also as a technology that facilitates the implementation of bridges between these spaces because *“Model Driven Development and Engineering with transformation approaches and tools can automate the generation of the corresponding target Space instances and domains”*.

Software Language Engineering

Interestingly, in recent years, increasing research effort has been directed at studying the relationship between precisely the two spaces which are of relevance in the

[3]In the context of model-driven engineering model transformations (M2M or M2T) can act as technology bridges.

context of this thesis, model-driven technologies and formal languages[4]. These areas are often referred to as *modelware* and *grammarware* by their respective community. In [Bra09], the close ties between both areas are ascribed to a common goal, *"raising the level of abstraction in software development"*. It is further noted that *"the lessons learned by [the generic language] community, can be beneficial for the model-driven engineering community"*. The field that bridges the gap between modelware and grammarware is known as software language engineering (SLE) [Kle09]

On the official website of the SLE conference[5], the following definition of this term is given:

Definition 4.1.2

> *"Software language engineering is the application of systematic, disciplined, and quantifiable approaches to the development (design, implementation, testing, deployment), use, and maintenance (evolution, recovery, and retirement) of these languages. Of special interest are (1) formal descriptions of languages that are used to design or generate language-based tools and (2) methods and tools for managing such descriptions, including modularization, refactoring, refinement, composition, versioning, co-evolution, recovery, and analysis."*

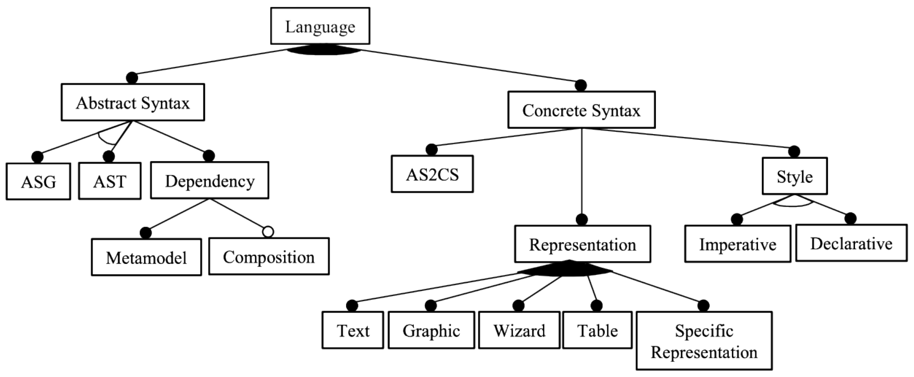


Figure 4.2.: Language features [LJJ07].

The similarities between both spaces are visualized in the feature diagram in Figure 4.2. It demonstrates that certain characteristics that constitute a language are actually interchangeable. A language definition in model- as well as in grammarware has to describe the available constructs (*the alphabet*) and how they can be arranged to form valid words or sentences in an *abstract syntax*. This can be accomplished using a variety of different techniques such as grammars and metamodels. The definition of the structural composition of a language has to be complemented by a specification of one or more visual representations for each language construct

[4](Semi) automatic bridges between those areas have been proposed in [AP04; Kun08].

[5]In 2008, an annual scientific conference named Software Language Engineering (`http://planet-sl.org`) has been established with the stated goal of exploring and strengthening the ties between modelware and grammarware.

(*concrete syntax*)[6]. Not shown in this diagram are the *static semantics* - constraints that cannot be expressed using the abstract syntax - as well as the *dynamic semantics* which state how language expressions are to be interpreted. Nevertheless, they constitute important parts of any language specification.

Conclusions

The premise for the adaption of the DFA technique to the field of modeling lies in the conceptual similarities between the spaces of modelware and grammarware. These similarities relate to the fundamental properties of both areas as well as to how the respective methods are used in practice. The following sections will therefore examine these categories by juxtaposing the technological spaces to provide a sound basis for the subsequent conclusions in Section 4.2. These, in turn, are a prerequisite for the actual implementation of a MDE-driven DFA approach for models. From Definition 4.1.2 we can conclude that this approach is firmly rooted in the field of software language engineering.

4.1.2. Application Domains

We now explore the conceptual similarities between modelware and grammarware by examining their usage in traditional as well as in newly emerging application scenarios. This study will help to identify relevant properties in both areas and assist in understanding how compiler construction techniques can be adapted to the target domain. Additionally, knowledge about the usage patterns is also beneficial for the task of facilitating a technical integration of the two language engineering approaches.

Modeling

While in the beginning, models have often been used for documentation purposes only, the discipline of modeling has quickly evolved into a powerful technology that is widely applied in different areas of software engineering. As [Bez05] states, *"[o]ne important difference between the old modeling practices and modern MDE is that the new vision is not to use models only as simple documentation but as formal input/output for computer-based tools implementing precise operations"*. Starting with the automated generation of code skeletons, more sophisticated methods soon became available such as the description of dynamic behavior (and subsequently the automated generation of fully functional programs). The application of MDE techniques forms the core of model-driven software development processes such as the Rational Unified Process (RUP) or the Model-driven Architecture.

A more recent development however, is the shift away from using predefined modeling languages to the specification of customized languages - called domain-specific languages - for specific purposes. These are *"custom- and purpose-built languages*

[6]The context-free grammars used to specify programming languages combine the definition of the language's abstract and concrete syntax.

that target a specific domain", *"often smaller than general-purpose languages (GPLs), providing fewer and more focused language constructs and, ideally, a simpler and more rigorous semantics"* with the intent to *"provide a concise, tailored language that is easier for engineers and domain experts to learn, understand and apply for a specific class of problem"* [Zsc+10].

This trend started with the realization that general purpose languages such as the Unified Modeling Language are overloaded with seldom required constructs and functionality which hinders their efficient usage by introducing a high level of complexity. At the same time, facilities that enable customization, such as UML profiles, are often limited in their expressiveness while at the same time being difficult to use. With the introduction of the MOF and accompanying tool support, it became possible for software engineers to build their own modeling languages with comparatively little effort, a practice that is now sometimes called domain-specific modeling (DSM). In this context, the authors of [Bez05] observe that current tools *"operate on top of a model and metamodel repository"* where the *"UML metamodel, which was previously at the center of these workbenches, is now only one metamodel among others"*. They predict that *"The MDA landscape is going to be populated by a high number of metamodels, like the programming language technical space which is populated by a high number of language grammars or the XML document space populated by DTDs and XML schemas"*.

With the rise of DSM came the expansion into new application areas. While the typical use case for UML is the specification of structural and behavioral aspects of software systems, custom modeling languages are able to target arbitrary domains. Once a language has been devised, MDE techniques can then be used to (semi) automatically supply the corresponding development tools.

The Java Workflow Tooling project (which is the basis of the case study in Section 10.1) is an example for the inherent advantages of the domain-driven approach. JWT supported the model-based definition of executable business processes several years before the OMG provided a well-founded specification of the BPMN[7]. For this purpose, JWT defines a DSM in the form of a precise metamodel which is strictly limited to a set of essential constructs. These elements enable the specification of executable business processes in a MDA-inspired notion by including platform independent (PIM) and platform specific (PSM) properties. A graphical business process editor provides multiple views on the modeled artifacts which resemble UML Activity Diagrams, BPMN processes or event-process chains (EPC) and can also be customized for different user roles such as business or technical experts[8]. Since all of these functions operate on the same data structures and underlying technical principles as defined and implemented by the language's metamodel, all changes are automatically synchronized independently of the chosen representation. JWT's IDE, which consists of graphical and textual editors as well as transformations to languages such as the BPEL (Business Process Execution Language) or XPDL (XML

[7]It can be argued, that the BPMN is inherently too complex to be used efficiently by business experts in the same way that the UML is for software engineers.

[8]In business process modeling (BPM), processes are usually defined by business experts (PIM) and then refined for execution on a specific process engine by technical experts (PSM).

Process Definition Language), has been implemented with comparatively little effort using MDE technologies.

Compiler Construction

As stated earlier, the usage of formal languages in software engineering can be traced back to the 1950s [Bac+57]. The field of compiler construction encompasses techniques that employ the theory of formal languages to parse expressions written in a source language and transform them to semantically equivalent terms in a target language. Since then, many improvements have been made in the development of compiler front-ends, e.g. regarding parser efficiency and fault tolerance. In the context of this thesis however, one the most notable extensions has been the introduction of attribute grammars to validate static semantics. Nevertheless, the underlying principles in this area have largely remained the same. This is evident from the fact that context-free grammars continue to form the basis for modern languages such as ECMAScript [Ass11] from which the popular JavaScript language is derived.

A more striking change on a practical rather than on a fundamental level can be found in the advent of the trend towards (textual) domain-specific languages that has intensified in the last years. In academia and industry alike, DSLs more and more replace general-purpose languages (GPL) as they provide non-technical users with simpler, yet at the same time more expressive, tools for completing tasks in their respective domain.

A canonical definition of DSLs can be found in [DKV00]:

Definition 4.1.3

A domain-specific language (DSL) is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain.

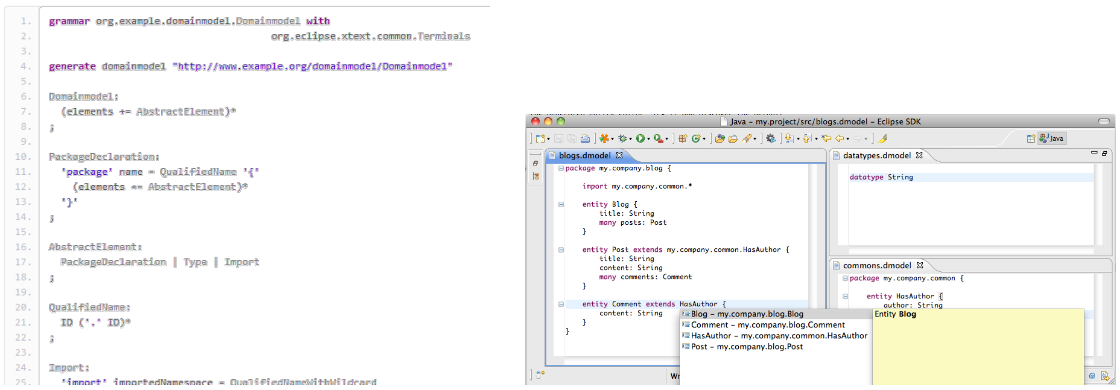
The authors further attribute the following characteristics to domain-specific languages which hold true in many - though not all - cases:

Limited size Usually, the set of language constructs is limited to the essential elements of a very specific domain.

Declarative specification Rather than for actual programming, DSLs are often used for more high-level specifications.

Engineering Domain-specific Languages

The trend towards domain-specific languages drastically changes the priorities of many aspects in established compiler construction techniques. In the past, developers relied on a given set of languages designed and provided by experts who are heavily entrenched in the details of formal language theory and compiler construction technology. Today, language engineering itself has become a field that is accessible to a growing target audience of developers. Thereby, a new challenge arises: To allow a wider array of people with different backgrounds to participate in this field, it is necessary to provide suitable techniques that support common requirements of the IT industry such as standardized development methodologies, interoperability and mature tooling support. At the same time, the emphasis traditionally put on aspects such as performance is more and more neglected in favor of the aforementioned goals.



```
grammar org.example.domainmodel.Domainmodel with
                                      org.eclipse.xtext.common.Terminals

generate domainmodel "http://www.example.org/domainmodel/Domainmodel"

Domainmodel:
  (elements += AbstractElement)*
;

PackageDeclaration:
  'package' name = QualifiedName '{'
    (elements += AbstractElement)*
  '}'
;

AbstractElement:
  PackageDeclaration | Type | Import
;

QualifiedName:
  ID ('.' ID)*
;

Import:
  'import' importedNamespace = QualifiedNameWithWildcard
```

Java – my.project/src/blogs.dmodel – Eclipse SDK

blogs.dmodel

```
package my.company.blog {

    import my.company.common.*

    entity Blog {
        title: String
        many posts: Post
    }

    entity Post extends my.company.common.HasAuthor {
        title: String
        content: String
        many comments: Comment
    }

    entity Comment extends HasAuthor {
        content: String
    }
}
```

datatypes.dmodel

```
datatype String
```

commons.dmodel

```
package my.company.common {

    entity HasAuthor {
        author: String
```

```
Blog – my.company.blog.Blog
Comment – my.company.blog.Comment
HasAuthor – my.company.common.HasAuthor
Post – my.company.blog.Post
```

Entity **Blog**

(a) Xtext context-free grammar

(b) Generated language toolkit

Figure 4.3.: Xtext as an example of the fusion of language engineering approaches stemming from both compiler construction and modeling.

As *"MDE is more and more related to DSL engineering"*, the authors of [Kur+06] suggest that *"MDE principles and tools may be considered as a convenient support technique for building DSL frameworks that may solve existing and newly emerging complex problems"*. A perfect example for this new development is the Xtext framework (cf. Section 7.1.4), a language engineering workbench for the Eclipse platform. Xtext not only bridges MDE and compiler construction by associating grammatical symbols with metamodel classes (cf. Figure 4.3(a)) but itself also heavily relies on model-driven engineering principles such as Model to Text (M2T) code generation[9]. Xtext can therefore be classified as a practical application of the principles of software language engineering.

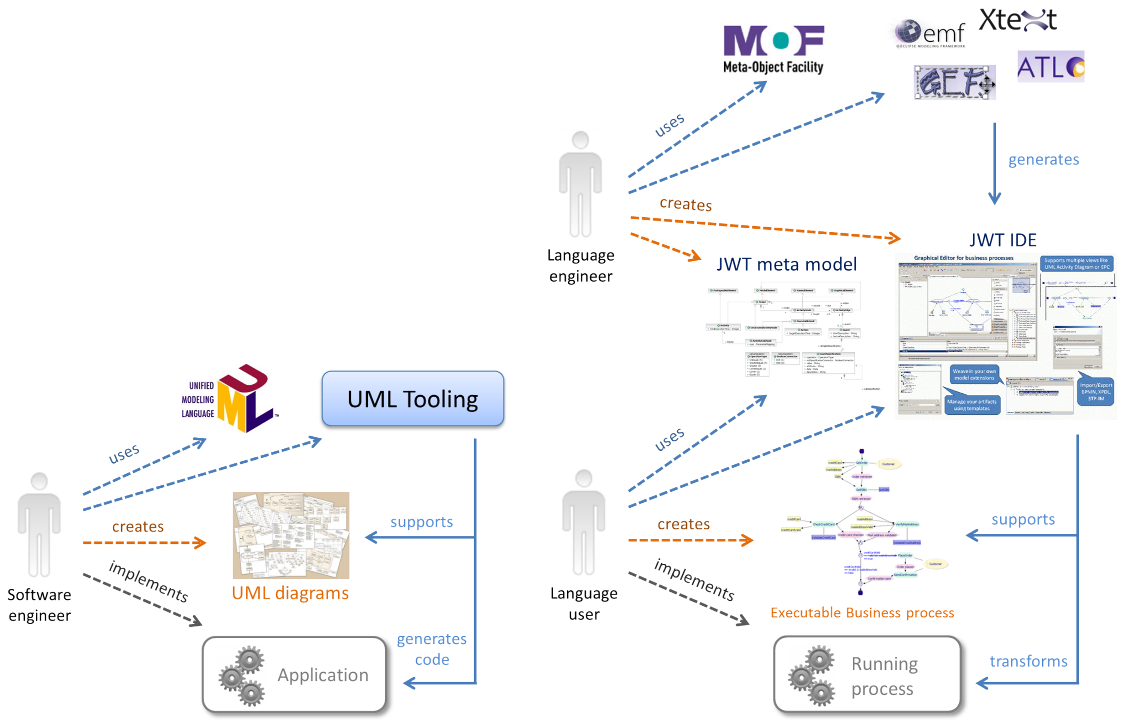
Although the IDEs generated by the Xtext language workbench (cf. Figure 4.3(b)) are considerably slower than a speed-optimized parser, it nevertheless proves to be a highly valuable and increasingly popular technology. Many features such as auto-completion or syntax-highlighting are supported out-of-the-box and can be adapted easily to project-specific needs. Also, the creation of DSL parsers and editors by means of code generation results in fast development cycles and prevents errors that often occur in manual implementations. This way, even developers with little expertise in language engineering are able to quickly achieve high quality results.

[9]This is also evident from the fact that Xtext has been used to bootstrap its own textual grammar editor. On a conceptual level, this can be compared to the self-hosting property of compilers that is often considered to be an important step in a compiler's development.

In [MH05], the authors of the OMG's Human-Usable Textual Notation (HUTN) state that this technique provides the ability to *"[build] models which conform to given metamodels, in a textual way [. . .] by specifying DSLs under the shape of metamodels [. . .] to generate basic textual tools to work with these DSLs"*. Since HUTN acts as a bridge between metamodels and parsers, it is therefore another example for the convergence of the trend towards domain-specific languages in the areas of modeling and compiler construction. In the paper, its benefits are summarized as follows:

- It is a generic specification that can provide a concrete HUTN language for any MOF model
- The HUTN languages can be fully automated for both production and parsing
- The HUTN languages are designed to conform to human-usability criteria

Conclusions



(a) UML-driven software engineering (b) Language engineering and usage

Figure 4.4.: Illustration of language use vs. language engineering exemplified by the application scenarios of UML and JWT respectively.

In summary, it can be concluded that there has been a major shift in recent years in the areas of modeling as well as in compiler construction. Interestingly, in both domains, the focus increasingly turns towards the development of domain-specific languages (DSLs and DSMs respectively) and the provisioning of corresponding IDEs. In contrast to existing general purpose languages such as programming languages or the UML, which are mainly used for software engineering purposes, the technologies available in both areas are often employed to target specific domains. This can be explained by a reduction of complexity that results in a more streamlined development process. Additionally, solutions that are tailored to a single domain or task tend to be easier to use and are therefore more accessible to business experts who are now able to formally specify development artifacts using familiar vocabulary. The ability to create languages which are specifically designed for clear cut tasks also means that developers don't have to rely on predefined standards which may not suit the requirements of the respective project.

The field of SLE establishes links between the areas of modelware and grammarware and thereby increased the popularity of the respective techniques by facilitating their usage outside their traditional areas of application. Xtext/HUTN and JWT were presented as examples with Xtext having a stronger focus on textual languages and JWT more aimed at the paradigm of graphical modeling.

The difference between the "traditional" application of the respective techniques and their usage in the notion of language engineering is illustrated in Figure 4.4. Figure 4.4(a) depicts how software engineers (architects, designers, programmers, ...) use predefined languages such as UML to specify the features of computer programs. MDE-based methods such as code generation then aid in the actual implementation of these systems. In contrast, Figure 4.4(b) focuses on the domain-driven approach employed, for example, by the Java Workflow Tooling (JWT) framework: Language engineers can resort to the rich ecosystem of existing (MOF-based) standards and tools to implement a model-driven tooling environment which is centered on a common metamodel. Since the resulting IDE is tailored to the properties of the respective domain, functionality is accessible by non-technical experts. In the case of JWT, usage comprises the design and implementation of executable business processes.

To distinguish between the different roles in the context of SLE, we introduce the following definition:

Definition 4.1.4

Language definition *(or language engineering/specification) is the task of employing SLE-enabled standards and tools to design and implement domain-specific language environments for language users.*

Language usage *refers to the process of using (generated) DSL environments provided by language engineers to a achieve a specific goal in the target domain.*

As the emerging field of software language engineering blurs the line between CC and MDE, the need for more advanced methods for validating modeling artifacts becomes apparent. Suitable techniques, such as attribute grammars and data-flow analysis already exist in the context of compiler construction. It can therefore be deduced that an adaption of these techniques to the modeling domain would clearly benefit the notion of SLE. To achieve this goal, any suitable methodology for model analysis has to be derived from an alignment of the concepts of model- and grammarware. Considering the two roles presented in Definition 4.1.4, any such method has to take into account the steps of language engineering as well as language use. More specifically, on the language definition layer, it is necessary to extend the existing modeling standards used in the target domain with artifacts for analysis specification. On the level of language usage, facilities must be provided that allow users to execute the analyses and interpret the results.

4.1.3. Alignment of Abstraction Layers

The concept of languages carries with it the inherent property of multiple layers of abstraction (or meta layers). By definition, two layers of abstraction are involved in the development and the usage of a language: One represents the language's specification and the other the set of language expressions (or language instances) conforming to this definition.

However, the classification of definition and an instance layers - or metamodel and model layers respectively - depends on the respective point of view. In computer science, there has always been a strong feeling that a language which has been constructed for the purpose of formalizing expressions should itself be based on a formalized description to provide a consistent and reliable framework[10]. It is therefore required that each language definition is guarded by another layer of abstraction on top of it. The problem of an endless stack of layers is typically resolved by the top-most (meta language) being reflective, i.e. supporting its own definition. As a consequence, depending on the current viewpoint, each layer is an *instance* of the layer above it (or in the case of the top layer, of itself). At the same time, each layer provides a language definition that supports the expressions on the adjacent level below (with the exception of the bottom layer) according to the respective instantiation semantics. This is an essential point as language developers and language users typically have different notions of what is considered to be the definition and what represents the instance layer which may easily lead to confusion.

The most common architecture of a language framework consists of four layers with the bottom-most level representing "real-world" objects. However, the constituents of this instance layer are highly dependent on the application domain and the employed technology. For example, in the diagrams shown in Figure 4.4, instances would correspond to executions of the developed application and running business processes respectively. Because this layer itself is a result rather than an artifact of the language development process, it is often omitted from representations of the relevant abstraction layers.

Because the structural layout of language constructs is not concerned with the

[10]This is evidenced by the ongoing efforts to complement the MOF, UML, OCL and others with rigorous specifications for both syntax and semantics, e.g. [RG98; CK04b; SZ08].

dynamic semantics[11], we can restrict our study of the alignment of abstraction layers to the abstract and concrete syntax (cf. Section 4.1.4) as well as the static semantics (cf. Section 4.1.5). In addition, we will examine how languages can be annotated with analysis specifications and how these extensions relate to language definitions on a conceptual level.

Because of the widespread use of MOF-based languages in the modeling domain and EBNF in the field of compiler construction and because of their relevance to the contents of the following chapters, we will base our following comparison of similarities and differences in the technological spaces of modelware and grammarware on these two techniques.

Four-layered architecture

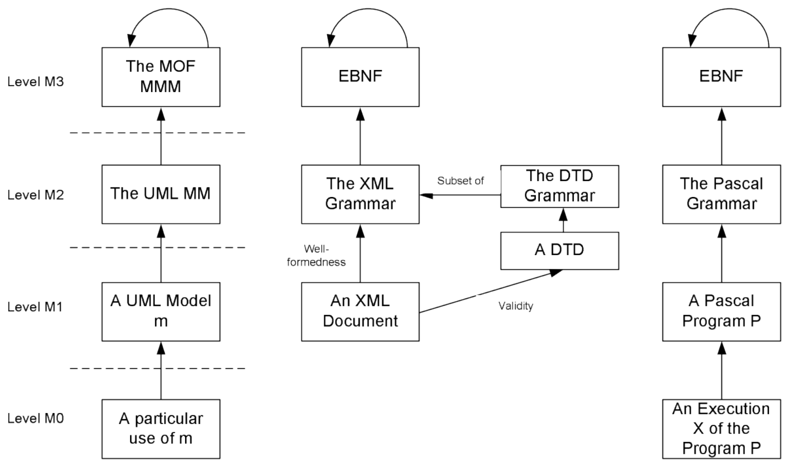


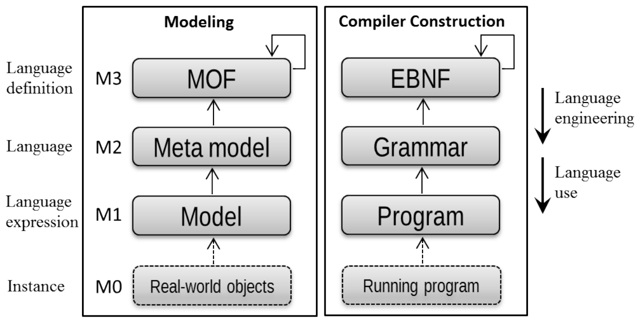
Figure 4.5.: Alignment of abstraction layers [KBA02].

Many researchers in the field of software language engineering stress the importance of interoperability between the technological spaces of modelware and grammarware [KBA02; WK05; AP04; Bra09]. This demand is not surprising as both areas play an important role in software development, albeit with very different fields of application. To achieve this goal, the authors propose technological bridges that are able to create mappings between language constructs. For this purpose, it is necessary to derive alignments for the respective abstraction layers.

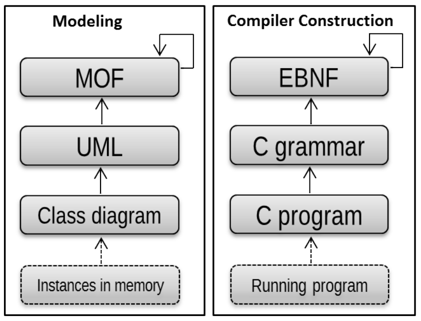
The consensus in the research community regarding the relationships between the different layers is illustrated in Figure 4.5. This diagram depicts the abstraction layers commonly associated with the technological spaces of modelware and grammarware, using MOF and EBNF as concrete examples for technologies in the respective area. As is evident from the arrangement, each layer has a unique counterpart in the other technological space. A placement on the same tier implies similarity both on a conceptual level as well as concerning the role it plays in the

[11] The reason for this is that restrictions imposed by dynamic semantics can only be validated during runtime.

use cases of language definition vs. language use[12].



(a) Generic form of the four abstraction layers



(b) Instances of the language stack for class diagrams and C programs.

Figure 4.6.: Alignment of abstraction layers in modeling (MOF) and compiler construction (context-free languages) with instanceof relationships.

We will now examine the areas of modeling and compiler construction in more detail. The architecture depicted in Figure 4.6(a) juxtaposes the language frameworks which use MOF and EBNF as their respective meta language. For further clarification, Figure 4.6(b) provides concrete examples for language (expressions) on each of the abstraction layers.

This arrangement of abstraction layers is *"the corner stone for building the model management functionality in a given space. It is mainly based on the fixed metametamodel at* M3 *and the meaning of the conformsTo relation between levels"* [B+05]. Although the *M3-M0* terminology stems from the modeling domain, we will therefore also apply it to the area of compiler construction.

Based on these observations, we extend Definition 4.1.4 with respect to the architecture shown in Figure 4.6(a):

Corollary 4.1.5

In the context of a four-layered language framework, we state that

Language definition *is the process of employing the* M3 *(meta language) layer to define* M2 *languages.*

Language use *refers to the usage of* M2 *layer languages to create language expressions which reside on* M1.

We will now address each layer individually, discussing its properties with respect to the process of language definition:

[12]The four tier architecture applies to other language frameworks as well, e.g. the KM3 [JBT06] which shares many properties with MOF. Another example would be the XML where a XML schema for schemas, XML schemas and documents correspond to M3-M1 accordingly.

M3 The top layer of a language framework is responsible for providing the generic facilities for language development. To be applicable in many different contexts, *M3* meta languages must possess a large amount of flexibility and expressiveness. Therefore, their constituents - including, amongst other elements, classes and nonterminals - are specific only to the respective technological space but not to any concrete application domain. For this reason, *M3* specifies only a small number of highly versatile constructs and concepts (such as generalization, references, derivation rules, ...). For this reason, both the MOF and EBNF often appear very compact in relation to some *M2* languages which may be of a much higher complexity. This is especially true if compared to general-purpose languages such as the UML or programming languages.

A common property of the top tier of a language framework is its ability for reflective description. Both the MOF's *M3* and the Extended Backus-Naur Form provide all constructs which are necessary for a specification of their own syntax.

M2 The language definition layer, usually referred to as *M2*, is located below the meta language. Languages on *M2* can be categorized along the lines of general-purpose and domain-specific languages, the most prominent example of a general purpose language in the modeling domain being the Unified Modeling Language. The respective counterpart in the area of formal languages would be a generically applicable programming language such as C or Java. BPMN or SQL, on the other hand, are domain-specific as they define a limited set of language constructs for concrete applications scenarios - the design of business processes and the specification of data-base queries. The SQL database query language is also an example of a textual DSL.

However, in a sense, the *M2* layer can always be considered to be domain-specific as *M2* languages, in contrast to *M3*, fulfill the requirements of a specific application scenario.

M1 Models and syntax trees, which represent the *M1* layer, are language expressions which are structured in accordance to their defining metamodel or grammar. In the case of UML, this could, for example, be a Class or an Activity Diagram while, for a textual language, it might be a program that has been processed by a corresponding parser[13]. *M1* therefore is the level on which language users operate by applying the language's elements with respect to the defined semantics. As mentioned before, in the hierarchy of abstraction layers, *M1* is also usually the last one of interest in the context of a language framework.

M0 As has been stated, the "real-world" instances on *M0* are usually not part of the processes of language definition or use. Overall, the whole engineering process is directed at facilitating the realization of the *M0* layer by providing suitable definitions and tools. Because the elements on *M0* are highly dependent on the used

[13]One could argue that, from a user's perspective, the relevant artifacts of *M1* would be the textual or the graphical representations rather than the parsed syntax trees or models. However, this distinction only concerns the mode of presentation, i.e the chosen concrete syntax.

technology, e.g. they way how objects are stored in memory or data bases, it does not fit in with the generic, technology-independent paradigm of the language-driven approach. In this context, *M0* therefore represents the results of the application of the language's facilities rather than being part of the modeling process itself. This view is supported by [B+05], in which the author considers "M0 *as being outside of the OMG TS*".

Mappings between Models and Grammars

After identifying the corresponding pairs of abstraction layers, it is essential to evaluate how their contents relate to each other. These mappings will provide a valuable insight into the challenges that arise when transferring the DFA-related concepts from one technological spaces to another.

An initial realization concerns the expressiveness of EBNF vs. MOF: As noted by [AP04], *"describing a mapping from metamodels to EBNF grammars is in many ways more demanding than the opposite"* because *"metamodels inherently contain more information than EBNF grammars"*. Therefore, arbitrary metamodels cannot be automatically converted to grammars while fully preserving their original expressiveness and semantics. On the other hand, mappings from EBNF to MOF exist, as described, for example, in [WK05]. As a result, the author of [AP04] comes to the conclusion that *"there is no direct isomorphism between any metaclass in an arbitrary metamodel and a token in a grammar"*.

Conclusions

We have presented an alignment between the abstraction layers in the fields of *modelware* and *grammarware*. Although the respective constructs cannot be mapped directly between both spaces, there are many conceptual similarities. Most importantly, this pertains to the notion of abstraction layers which govern the instantiation of language constructs and their application in the tasks of language definition and language use.

Due to unique properties of each technological space, for example a model possessing an inherent graph structure as opposed to a syntax tree, it is clear that these issues must be identified and addressed by any technology bridging these two spaces. This will be investigated in more detail in the next sections.

As a consequence of the alignment of abstraction layers, we can also conclude that any approach intended to extend modeling languages with compiler construction techniques such as data-flow analysis has to take into consideration all of the relevant layers in both fields, that is *M3*-*M1*, and their relationships with each other. Essentially, this is required because an analysis performed on objects contained in *M1* has to be defined on the *M2* level alongside the metamodel's elements. For this purpose, it should preferably employ the same basic constructs on which the metamodel itself is built, i.e. elements of the Meta Object Facility.

4.1.4. Comparison of Syntactical Concepts

Languages codify a set of instructions or descriptions that carry a specific meaning in a given domain. Thereby, they provide a standardized interface through which multiple parties can communicate information in a certain domain (DSL) or through a certain technology (GPL). For formalized languages, this interface exists either between computers or between computers and humans. For inter-machine communication, software systems are typically built to operate directly on the structure given by the abstract syntax as it is made accessible through a technological interface. For example, programs can easily operate on serialized XML data or model graphs without the need for a human-readable representation. Interactions between computers and humans, on the other hand, normally require a more descriptive representation of the structured information. This usually involves both filtering and enhancing the language expressions to include domain-specific aspects in order to make the syntax more accessible to human readers. For example, the visualization of a UML Class aggregates information about a subgraph of the underlying model consisting of the class itself as well as associated attributes and operations. This differentiation between the structural composition and its representation is a prerequisite for complementing a language's abstract syntax with one or more concrete syntaxes.

This concept will also play an important role in the definition of a data-flow analysis technique that can be applied to models. Not only is it necessary to provide a method for annotating a modeling language with flow analysis specifications, but the DFA's elements itself must be linked to a concrete representation that facilitates the specification language's usage. In this section, we will investigate how the abstract and concrete syntaxes of the technological spaces in question relate to each other. From this, we will draw conclusions about the properties of a domain-specific language which enables the definition of data-flow analyses.

Concepts of Abstract and Concrete Syntax

Arguably the most obvious difference between modelware and grammarware is that the former is commonly associated with graphs while in the latter case textual representations are used. This perception can however be somewhat misleading, since the abstract representation of a context-free language expression, a syntax tree, is in fact also a graph, albeit with certain restrictions on its structural composition. More specifically, as can be deduced from their name, syntax trees always possess the form of a tree. The nodes represent grammatical symbols while edges denote the application of derivation rules. However, although the distinction between both representations is getting more and more blurred by the advancing field of SLE - e.g. in the form of Xtext (cf. Section 4.1.2) which assigns a textual syntax to modeling languages - it still holds true for many applications.

Table 4.1 lists the abstract and concrete representations for the compiler construction language stack. The EBNF specification contains both the expressions required to formalize a grammar as well as a reflective definition of the EBNF using its own syntax. While the concrete representation is always in a textual format, the internal

Layer	*Artifact*	*Abstract Syntax*	*Concrete Syntax*
M3	EBNF	AST of EBNF grammar	Textual repr. of EBNF grammar
M2	Grammar	AST of grammar	Textual repr. of a grammar
M1	Program	AST of program	Textual repr. of a program

Table 4.1.: Abstract and concrete syntax representations of the CFL framework as used in compiler construction.

structure is that of a syntax tree.

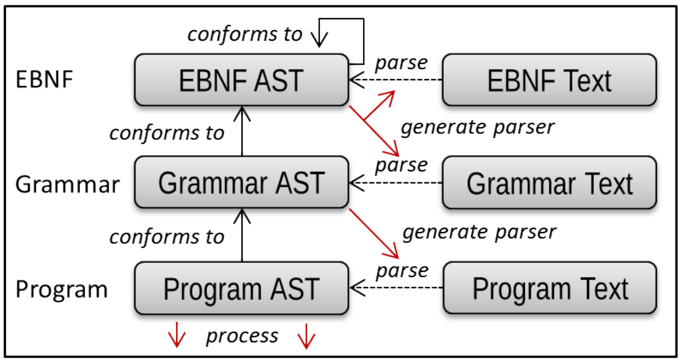


Figure 4.7.: Relationships between abstract and concrete syntax in CC.

This relationship is further detailed in Figure 4.7. In this diagram, it can be seen that the concrete syntax is only an additional feature through which one attempts to improve the readability for human users while the actual syntactical structure is encoded in the syntax tree. Based on the properties of context-free languages, algorithms can be constructed from the language's definition that parse the concrete representation and reconstruct its structure according to the abstract syntax. It is important to note that the *M3* and the *M2* layer conform to exactly the same specification with respect to both their abstract and their concrete representations. This means that the textual representation of EBNF and that of a grammar defined using EBNF employ the same syntactical elements to describe terminals, non-terminals and derivation rules. Therefore, a parser programmed to interpret EBNF constructs can also read grammars. In contrast, parsing programs based on user-defined grammars requires a separate logic that is able to interpret the constructs defined on *M2*. Since EBNF grammars already encode the concrete syntax in their terminal rules, the generation of language parsers can be automated. However, because the symbols and their structural arrangement on *M2* encode domain-specific information, the contents of *M1* syntax trees are domain-specific as well. Therefore, further processing of a program's AST, e.g. by subjugating it to static analysis or transforming it into machine code, has to be implemented manually for each defined language.

Table 4.2 focuses on the abstract and concrete representations found in the modeling domain. From an examination of their properties, it is evident that the relationships between them are in fact very similar to the technological space of compiler construction: Again, both the *M3* and *M2* layers share the same properties regarding their syntactical elements and structural composition. Any viewer or editor that

Layer	*Artifact*	*Abstract Syntax*	*Concrete Syntax*
M3	MOF	Model of the MOF	Class diagram of the MOF
M2	Metamodel	Model of the metamodel	Class diagram of the metamodel
M1	Model	Model	Domain-specific representation

Table 4.2.: Abstract and concrete syntax representations of the MOF.

is able to interpret MOF-based metamodels or operates on their abstract syntax will also be able to process MOF models (i.e. metamodels).

In contrast to the EBNF, the MOF language however does not provide any means to assign concrete representations to their abstract counterparts. Therefore, to serialize, parse and visualize models, additional techniques are required. A wide variety of methods exist to fill this gap, ranging from XML-based serialization (XMI) to textual representations (Xtext) and diagram editors (GMF). Because the constructs defined on *M2* are domain-specific, it is difficult to automate the process of supplying visualizations for model elements. For example, diagram-based representations of models typically distinguish between nodes and edges. In this case, the information which abstract concept should be visualized as a node and which conforms to an edge cannot be automatically derived from the metamodel and thus has to be supplied by the language engineer.

We will now list the most significant differences between the two abstract representations with respect to our use case:

Graph vs Tree Structure
Parsed expressions based on an EBNF grammar form a syntax *tree* where each node conforms to a (non)terminal and each edge to a grammatical derivation. In contrast, while the containment hierarchy of (meta) models forms the structural backbone (which is, for example, used by the XMI serialization), non-containment references result in cross-reference edges which have no equivalent in syntax trees. Therefore models form (directed) *graphs*.

Edge semantics
In syntax trees, edges always denote the application of a derivation rule. In model graphs, edges are instances of references (generalization, containment, cross-reference) which in turn are characterized by certain properties (role names, navigability, multiplicity) which possess defined semantics that affect instantiation and interpretation on the model layer. However, modeling languages may also specify additional, domain-specific references between elements by defining classes that function as edges.

Element ordering
The ordering of elements enforced by grammars is not found in (meta) models. This aspect is required by the parsing algorithms to uniquely identify the correct derivations for the syntactical expressions.

Assignment of Concrete Syntax

EBNF languages implicitly assign concrete representation to the abstract syntax. This method not only automates the process of parsing textual representations of language expressions, a parsed expression's concrete syntax can also be reconstructed by traversing the syntax tree in depth-first order and printing the strings at the leaves (terminal symbols)[14]. Generally, graphical syntaxes are not well suited for representing syntax trees as their limited expressiveness does not warrant the inherently more complex depiction in a graphical representation.

Models, on the other hand, both allow and require a greater level of freedom when devising suitable visualizations. This makes the task of assigning a concrete syntax to language constructs more complex when compared to CFLs. A domain-specific visualization of a model graph may, for example, require the aggregation of multiple elements into a single representation (e.g. classes with their attributes and operations) and the visual style of nodes and edges may vary depending on their type. As a consequence, it is difficult to automatically derive a graphical syntax from a metamodel. The basic structure of textual mappings (e.g. based on Xtext), on the other hand, can often be generated using the containment hierarchy as guidance for element nesting. Nevertheless, these mappings often have to be refined manually.

In general, we can isolate several characteristic properties that govern the definition of a concrete syntax and its assignment to abstract concepts in modelware and grammarware:

Representation
: For both models and formal languages, there exists a dominant form of representation for the language constructs on all levels of abstraction. For programming languages, this is a textual format while for models (directed) graphs with customized representations for nodes and edges are usually employed. This choice is influenced by both the structural properties of the underlying formalization technique as well as by the requirements of the application scenarios in the respective technological space. However, in recent years textual DSLs have made a more prominent appearance in the modeling domain.

Binding/Assignment
: The method employed to define mappings between language elements and their concrete representations is a crucial aspect of the specification process. This may happen implicitly as is the case with EBNF which uses terminal rules to encode the actual expressions of the concrete syntax as well as their structural composition. As a consequence, editors and parsers can be generated automatically. To a lesser extent, this also applies to textual representations of models, albeit there are some limitations such as unnamed references which complicate this procedure. If, however, a graphical syntax is required, mappings have to be defined manually. Because the interpretation of the contents

[14] Because white space is normally omitted in the abstract representation, this is usually complemented by pretty-printing or auto-formatting.

of a model is highly domain-specific, the method for specifying these mapping depends on the chosen visualization technique.

Construction
It can be observed that model editors tend to synchronize the displayed diagrams with internal representations of the model's abstract syntax. This is in stark contrast to textual programming languages, where the syntax tree is constructed retrospectively. In other words, the concrete syntax of a programming language is regarded as the preferred method to encode language expressions while model diagrams are often treated as "syntactical sugar" that merely enhances the abstract representation. An important reason for the divergent approaches is that models usually cannot be reconstructed from their visual representation.

Extending the Meta Object Facility

Assigning a concrete representation to model elements is a complex task that usually requires manual engagement by the language engineer. The definition of a concrete syntax is however not the only instance in which mappings to model elements play an important role. It has already been mentioned, that a method for executing data-flow analyses on models not only requires the development of a suitable analysis specification language, but one also has to think about how these specifications can be linked to the underlying models and how these links can be represented.

As a constraint language for the modeling domain, the Object Constraint Language naturally has to deal with the same challenges: The typical use case consists of the annotation of constraints at metamodel elements. To support this procedure, links between expressions and target elements have to be encoded in the abstract syntax while the concrete syntax must define a corresponding representation.

Effectively, this approach involves layers *M3* to *M1* of the MDE language stack. The language specification of OCL basically acts as an extension of the meta language MOF. The actual OCL queries are language expressions conforming to the *M3* specification. They are defined on *M2* and connected to metamodel elements residing on the same abstraction layer. On *M1*, these expressions are instantiated alongside the modeling language's constructs and can then be evaluated by an OCL interpreter.

The extension of the *M3* layer with OCL's language definition is shown in Figure 4.8. It can be seen that these constructs define links to the respective MOF classes (depicted as transparent blocks). Since both metamodels and OCL constraints are *M2* artifacts, the creation of links between these elements is supported by the MDE paradigm.

It is worth mentioning that the language definition itself is designed in a way that facilitates mappings between OCL constructs and grammatical derivation rules and thereby supports a straightforward implementation of a textual syntax. Nevertheless, in practice, the visualization of constraints also involves a graphical component. While the constraints themselves are written using a textual syntax, they can be included in a graphical representation of the target metamodel. In this case, the

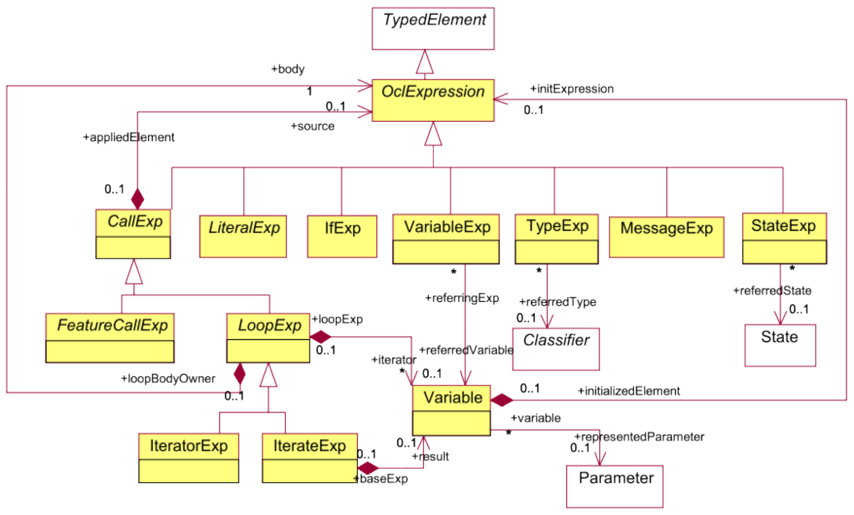
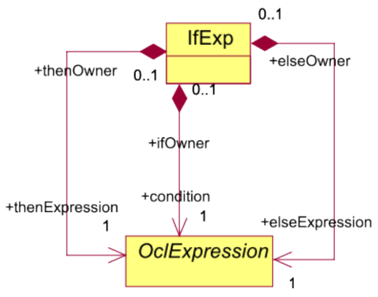


Figure 4.8.: Extending MOF: The OCL expressions package [OCL].

annotation relationship between a constraint and its context class is denoted by an edge that connects both elements. However, this requires that the respective metamodel tooling implements the OCL extensions on the *M3* layer to make the corresponding features available for the development of *M2* metamodels.



(a) Abstract syntax

```
IfExpCS ::= ‘if’ OclExpression[1]
          ‘then’ OclExpression[2]
          ‘else’ OclExpression[3]
          ‘endif’
```

Abstract syntax mapping

IfExpCS.ast : IfExp

Synthesized attributes

IfExpCS.ast.condition = OclExpression[1].ast
IfExpCS.ast.thenExpression = OclExpression[2].ast
IfExpCS.ast.elseExpression = OclExpression[3].ast

Inherited attributes

OclExpression[1].env = IfExpCS.env
OclExpression[2].env = IfExpCS.env
OclExpression[3].env = IfExpCS.env

(b) Concrete syntax

Figure 4.9.: Definition of abstract and concrete syntax for If expressions [OCL].

OCL connects the abstract syntax to a concrete representation using attribute grammars. As depicted in Figure 4.9, this method comprises:

- A grammar for parsing expressions (IfExpCS)
- An *abstract syntax mapping* that connects the model of an expression to a semantic attribute ast in the grammar that computes the abstract syntax tree
- An additional attribute env that determines variable visibility in the context of each (sub) expression

The Object Constraint Language therefore combines multiple aspects from the domains of modelware and grammarware to implement a constraint language for models by extending the MOF framework on layers *M3*-*M1*.

Conclusions

In this section, we studied the characteristics of context-free languages and modeling languages with respect to their abstract and concrete representations. Each domain has a preferred method of visualization that is used consistently throughout the abstraction layers of the respective language framework. Since both *M3* and *M2* are expressions of the same meta language, a common property of both areas is that these artifacts can be processed using the same tooling, e.g. an editor or a parser. The same is however not true for *M2* languages since their constructs are defined by the developer who subsequently has to specify how *M1* instances of these elements should be represented and interpreted.

For grammars written in EBNF, the concrete syntax is already encoded in the language's definition. This information can be used to automatically derive parsers capable of transforming the concrete textual representation into a syntax tree. To a lesser extent, this approach can also be applied to metamodels through techniques such as Xtext or HUTN. However, because of the more complex structure allowed by the modeling standards, this is not always possible. The provisioning of a graphical syntax, on the other hand, almost always has to be implemented manually by the language engineer. Based on an excerpt from the OCL's specification, we described how this technique manages to provide a textual constraint language that is nevertheless tightly integrated with the modeling domain. More specifically, we examined how an extension of the Meta Object Facility framework enables the definition of constraints and their annotation at metamodel elements on *M2*.

Since the Object Constraint Language can enrich the abstract syntax of modeling languages with automatically verifiable implementations of well-formedness rules, it is clear that this method has conceptual ties to the flow-based technique for model analysis that is the subject of this thesis. It is therefore possible to derive conclusions that are relevant to the definition of a syntax for model-based data-flow analysis from the presented techniques and approaches:

- To provide a consistent integration with the modeling domain, the abstract syntax of the analysis specification language should itself be governed by a metamodel.
- It should be possible to annotate analysis constructs at their respective counterparts in the target metamodel in a non-intrusive way.
- The different abstraction layers involved in the definition and usage of analysis expressions have to be aligned with the *M3*-*M1* layers of the MDE domain.
- Although the analysis operates on models, it should be representable in a textual format which can be processed by a parser.

4.1.5. Static Semantics in the Context of Language Engineering

In this section, we will examine how static semantics are used in the domains of modelware and grammarware. Based on these descriptions, we will derive requirements that must be implemented by the flow-based approach to model analysis.

Static Semantics the CC and MDE Language Frameworks

The technique of context-free languages provides a convenient way for defining both the abstract and the concrete syntax of a text-based language and enables the efficient parsing and interpretation of expressions. However, when relying solely on grammatical rules to specify the syntax, the ability to impose restrictions on what constitutes a valid language expression is very limited. As is apparent from the term "context-free", each partial expression is examined locally, neglecting the overall context in which it appears, i.e. the transitive closure of its parent and child expressions. This presents a problem in the design of programming languages because the usage of some language features may only be valid if certain preconditions are met. For example, in many cases, assigning a value to a variable requires that the variable has been explicitly declared before it is accessed and that the value that is assigned to the variable is of the correct type. While the information about which variables are available in the context of a given expression can be extracted from the syntax tree, context-free grammars do not provide the necessary facilities to encode and enforce this restriction. To compensate for this shortcoming, the abstract syntax must be complemented with additional specifications that are able to derive contextual information by taking into account the position in which each language element appears. The most commonly used technique for this purpose in the field of compiler construction are attribute grammars which operate directly on the abstract syntax tree (cf. Section 2.2). It can be argued that data-flow analysis also belongs to this category, since the program's control-flow graph is also an abstract representation of a program's syntactical structure, albeit further processed and refined.

In compiler construction, the term *static semantics* usually refers to the subset of restrictions that cannot be expressed using the abstract syntax but nevertheless can be validated statically. Compliance with these constraints can therefore be checked on an abstract representation of the program such as a syntax tree or a control-flow graph. [Ode93] states that *"traditionally, the term encompasses all aspects of language definition that fall between context-free syntax and semantics, and are thus concerned with neither sentence structure nor the meaning of programs"*. Correspondingly, methods such as attribute grammars constitute a static analysis. This classification is challenged by some in the formal language community (e.g. [SK95]) who argue that these requirements should instead be classified as syntactical rather than as semantical properties. Nevertheless, since *static semantics* is a well-established term in the field of compiler construction, we will subsequently adhere to this traditional interpretation.

In Section 4.1.3 and Section 4.1.4, we presented an alignment of the abstraction layers that are an integral part of the technological spaces of CC and MDE and compared their respective facilities for syntax specification. From this study, we can deduce that, because of the similarities between both language frameworks, the modeling methodology suffers from the same shortcomings as formal languages with respect to the formalization of their well-formedness rules and other static analyses. Indeed, the importance of complementing the UML specification with formalizations of its static semantics has been recognized at a very early stage of its development [EK99]. Generally, a model is considered to be syntactically valid if it adheres to the structural composition that is defined by its metamodel. However, just as is the case with context-free grammars, metamodels lack the ability to include constraints that take the overall context of model elements into consideration.

In [SB13], we noted that *"[o]ver time, existing formal approaches have been proposed for the purpose of model analysis"*, e.g. to implement dynamic model checking [SCH02]. *"However, this usually involves a translation of (meta) models into logic-based representations [MM06; SAB10] resulting in a gap between the two domains that can be difficult to manage on a technical level but may also lead to problems on a conceptual level as model-specific semantics have to be mapped to the logic-based systems on which the analyses are defined and executed"*.

The issue of providing a method for the static analysis of models has been addressed by the OMG's Object Constraint Language. It enables the annotation of constraints at metamodel elements and their evaluation on the instance layer. The authors of [B+05] classify OCL as a *navigation language* for the technological space of modeling which is *"associated to the basic representation system imposed by the metametamodel"* because *"a basic need when working with a model is to access model elements in a fine-grained manner"*. In addition to providing the means to address a certain subset of model elements through navigational expressions, OCL *"also serves as an assertion language and may be even used as a side-effect free programming language for making requests on models and meta-models"*.

With respect to the OCL, in [SB13], we stated that *"limitations of its expressiveness due to its static navigational expressions are the subject of ongoing discussion [MV99; Baa03]"*. One recent attempt at enhancing the expressiveness of OCL is the newly introduced closure() operator[15] which *"only applies to Set types and is limited to calculating the transitive closure of a relationship"*. We also noted that *"it has been argued that OCL itself lacks a proper formalization [BDW06] and multiple proposals have been made to address this problem [CK01; MB06; BW02]"*.

When examining the capabilities of contemporary static analysis techniques, it becomes clear that the area of modeling falls short when compared to the field of compiler construction with its well-established methods of attribute grammars and data-flow analysis. Both of these methods are inherently flow-sensitive, i.e. the computations carried out at a specific node in the syntax tree or the control-flow graph depend on the results at neighboring nodes which in turn are based on results calculated at adjacent elements. Apart from the newly introduced `closure()` operator with its limited applicability, OCL does not provide support for the analysis

[15] A typical use case can be found in the enforcement of non-cyclic generalization hierarchies for Classifiers: `self->closure(superClass)->excludes(self)` [SB13].

of elements based on their relative position in the model.

Static Analysis of Model and Context-Free Languages

As a prerequisite for the implementation of our approach, we will now motivate and describe the necessary requirements that must be met. To derive these requirements, we examine a conceptual alignment of techniques for static analysis in the technological spaces of compiler construction and MDE. For this study, we focus on how attribute grammar extensions relate to the artifacts of context-free languages in the scenarios of language definition and use. We then juxtapose these findings with the abstraction layers of the modeling domain and develop the requirements for the implementation of an equivalent, attribute-based static analysis technique for models.

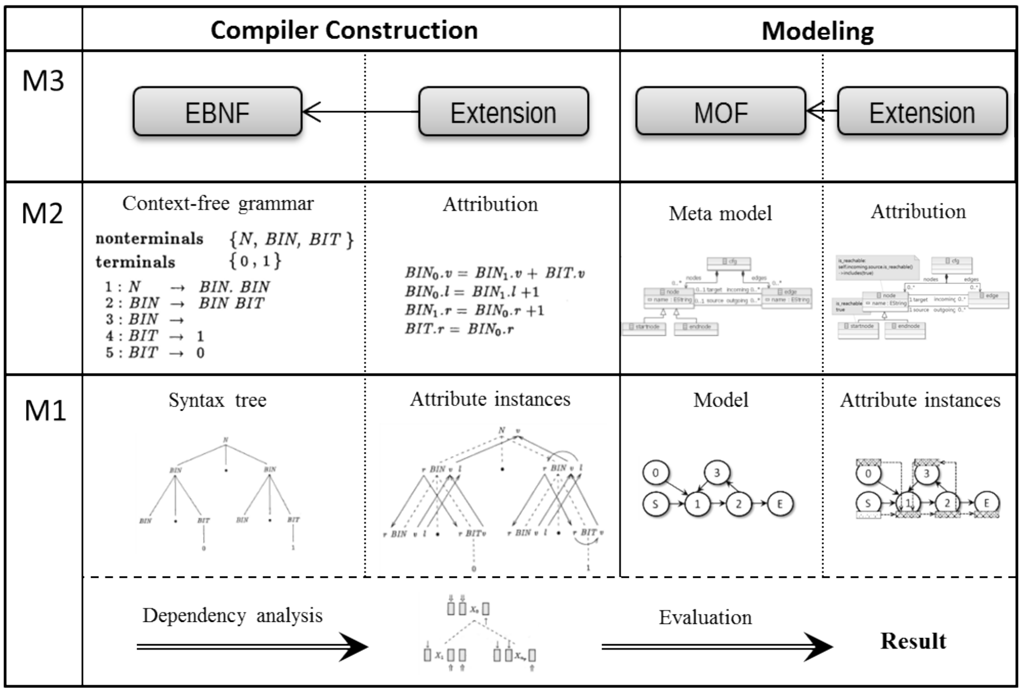


Figure 4.10.: Alignment of static analysis in CC and modeling.

Figure 4.10 depicts the relationships between the language and the analysis artifacts on the different layers of abstraction in both language frameworks. In general, by extending the meta language on *M3* with attribution concepts, it becomes possible to specify analyses on the *M2* layer which can then be instantiated for *M1*.

In the case of context-free languages (left hand side), the attribute grammar extension annotates (semantic) attributes at occurrences of grammatical symbols inside the production rules. On *M1*, these attributes can then be instantiated at the applications of production rules in the syntax tree. The concrete layout of the syntax tree determines the dependency relationships between the instances and thereby influences the information flow computed for this language expression.

An important property of this approach is that the definition and usage of analysis specifications can be separated from the artifacts of the target language on all layers. Thereby, the language framework can be enhanced with analysis capabilities without requiring changes to any existing definition. Furthermore, the syntax of the analysis specification language is formally specified using the techniques provided by the technological space. As a consequence, the language framework is extended with the ability to define and execute analyses in a consistent and non-intrusive way. Consequently, the right hand side of Figure 4.10 hints at the implementation of an equivalent technique for the modeling domain.

Conclusions

In this section, we compared the conceptual principles behind methods commonly employed for the static analysis of languages. The validation of static semantics extends the restrictions defined by the abstract syntax. These constraints can be checked statically but their evaluation requires the examination of elements in the context of their environment. In the technological space of modeling, the Object Constraint Language is often employed to enrich metamodels with semantic constraints. However, in contrast to attribute grammars and DFA, which are powerful methods for the analysis of syntax trees and control-flow graphs, OCL does not provide the ability to implement declarative, flow-sensitive analyses. Our objective therefore lies in the transfer of the capabilities already available in the field of compiler construction to the modeling domain.

In Section 4.1.3, we investigated the conceptual similarities between the spaces of modelware and grammarware. Based on these findings, we argue that the application of compiler construction methods to the modeling domain represents a feasible solution for the presented problems. Because of the similarities in the syntactical structure of context-free languages and models (cf. Section 4.1.4), attribute grammars in particular can be viewed as a sound basis for the specification of model analyses. The comparison of the syntactical properties of static analysis techniques shown in Figure 4.10 provides further clues about the relevant aspects that must be considered in this undertaking.

By enhancing metamodels with semantic attributes whose values are then computed using DFA fixed-point evaluation semantics, we can implement a generic, declarative method for computing static properties that can be derived from a model's syntactical structure. A major advantage of this approach is that it supports an implementation of transitive specifications with little effort. This is often required for the calculation of context-dependent properties. For example, the transitive closure of a node's parents can be specified as `allParents = directParent ∪ directParent.allParents`. This way, we can refine analyses through static approximations of dynamic behavior, e.g. by computing which nodes will be visited on all possible paths leading to an action in an UML Activity Diagram.

In summary, from a conceptual view point, the analysis of language constructs (models and syntax trees) requires analysis specification on the language level which has to be supported by appropriate constructs on the *M3*/language definition layer.

Semantic attributes in the notion of attribute grammars are assigned to language elements on *M2* using the concrete syntax of the attribution language. Based on the structure of *M1* language expressions, these attributes can then instantiated and evaluated according to defined semantics. This design is in accordance to the conclusions derived from the general alignment of abstraction layers in Section 4.1.3.

We conclude our discussion with a summary of the requirements that must be met by a suitable approach for flow-sensitive model analysis:

- The approach should extend the expressiveness of OCL with the ability to formalize the data-flow between language elements based on local, declarative specifications.
- The definition of analyses on *M2* should be supported by a model-based language specification that extends *M3*.
- To preserve compatibility with existing standards and tools, the process of analysis specification should be non-intrusive, i.e. it should not necessitate modifications of existing language artifacts.
- Corresponding instantiation and evaluation semantics must be provided to instantiate specified analyses for models and compute the results using fixed-point semantics.

4.2. Towards an Application of DFA to Models

In this section, we will describe the concrete requirements and the principal characteristics of our approach. These points will represent the foundation for the formal and technical specifications in Chapter 5 and Chapter 6.

In Section 4.1, we investigated the properties of the technological spaces of model-driven engineering and compiler construction and their relations in the context of software language engineering. Based on these results, Section 4.2.1 describes a set of design goals for the application of data-flow analysis to modeling languages and subsequently outlines the resulting challenges.

The specific properties of the MDE domain have implications on the listed goals and challenges. These are addressed in Section 4.2.2. Furthermore, we outline the basic principles of our approach and introduce the artifacts that are required to provide a consistent framework for analysis specification and execution. Finally, we describe the steps that have to be taken to implement these artifacts, taking into account the presented design goals and challenges.

4.2.1. Design Goals and Challenges

The discipline of software language engineering bridges the gap between the spaces of MDE and compiler construction on a conceptual as well as on a technical level. Any technique that follows the principles of SLE should therefore consider the implications of the specific properties of both domains. This not only pertains to the theoretical aspects of the underlying concepts of formal languages and metamodeling but also to the requirements of language usage. For this purpose, it is necessary to present the non-functional requirements that are relevant to the practical application of the technique. Additionally, the influence of these requirements on the actual implementation of the proposed technique must be examined.

Consequently, the first part of this section details the non-functional design goals while the second part deals with the technical challenges that arise from the implementation of these goals.

Design Goals

We will now motivate and describe the high-level, non-functional design goals that pertain to the formal and technical specifications of the analysis technique:

1. **Staying in the Modeling Domain**
   Performing an analysis on a model does not necessarily imply that the analysis itself has to be model-based. For example, the model's elements and their relations could be extracted and transferred into a formalism such as Alloy (cf. Section 3.3). However, employing the capabilities of the underlying language framework as a basis for analysis specification has the added benefit of a seamless integration both on a conceptual and on a technical level. More specifically, an integrated representation of modeling languages and associated analyses prevents a syntactical and semantical gap as definition, instantiation and interpretation of analyses follow the same principles that also apply to models. It also means that existing MDE facilities can be used to support the development process. Useful techniques in this context range from model transformations and code generation to tools such as Xtext that support the creation of IDEs for the engineering tasks of analysis specification and execution. Additionally, a model-based methodology makes it possible to include support for well-established standards like OCL and QVT which can then be used to formalize data-flow equations.

   Generally, this requirement reflects the approach of attribute grammars which, while being an extension of context-free grammars, are themselves implemented using the facilities of the compiler construction domain (cf. Section 4.1.5).

2. **Evolution instead of Revolution**
   In an area with many well-established standards and tools, it is usually preferable to extend rather than to reinvent. This means that any contribution to this field should be made in a way so that it complies with existing technologies and does not change any of the established notions or practices. It also means that any extension should be driven by minimalism, making use of available technologies where possible and introducing new concepts only when necessary. The obvious advantage of a minimalist approach is a reduction of the effort (and thereby also of the amount of error sources) that has to be put into the specification and the implementation of the technique. With respect

to the practical application, building upon existing standards also simplifies the usage by those already familiar with tools and standards in this field and simplifies a seamless integration with existing methodologies such as development processes. As a consequence, language engineers and tool builders only need to familiarize themselves with constructs that expand upon concepts that they are already well-acquainted with while language users are presented with ready-to-use solutions specifically tailored to their domain.

3. **Compatibility**
   The requirement of *compatibility* augments the previous goal, *evolution instead of revolution*. It specifically refers to the problems arising when introducing a new technique into an existing technological ecosystem. To provide a pragmatic solution, the analysis technique should be able to function purely as an extension that does not disrupt any established work-flows and can be either used or ignored according to the requirements of the current situation. Developers, as well as existing tool chains, should not be concerned with the technical implications of the application of a technology if their responsibilities in the development process do not require it. A strong focus therefore has to be put on preserving conceptual and technical compatibility with relevant standards, which - in the field of SLE - include the Meta Object Facility and the Object Constraint Language. This design goal is also relevant because standards and tools may change over time. Any proprietary modification of standards or tools increases future efforts as one would be forced to update the custom adaptions and depending implementations.

4. **Ease of Use**
   To be of practical use, any analysis methodology has to be easily accessible by users and software tools alike. DFA is arguably a more advanced method with which many software experts may not be as familiar as, for example, with the UML. However, it is reasonable to assume that an experienced software engineer who is already proficient in model validation techniques such as OCL will be able to develop an intuitive understanding for a flow-based approach to model analysis.

   To support the process of familiarization with this technique, an emphasis should therefore be put on making its usage as intuitive as possible. This can be achieved by providing facilities that enable users to focus on the actual task - encoding information flows between model elements - rather than having to deal with the technical details of the specification process or the subsequent evaluation phase. To accomplish this, modifications can be made to the DFA technique in the transition to the modeling space that simplify the definition of analyses by automating the handling of data dependencies. Furthermore, automated user-assistance functions should be provided in the form of a dedicated analysis specification DSL and accompanying tool support.

5. **Versatility**
   The *versatility* aspect refers to both the range of different application scenarios

to which the analysis method can be applied as well as to the versatility on a technical level.

Concerning the application scenarios of static analysis, two different use cases can be identified: On the one hand, static model analysis is often employed to validate language instances by computing a verdict that indicates whether a model adheres to the defined static semantics. In a broader sense, these constraints may also include a set of codified modeling guidelines or metrics that give an indication of the model's quality. On the other hand, static analysis can also be used to extract implicitly contained information that is required in subsequent phases of a development process[16]. In a sense, these two scenarios are very similar in nature because the validation process equates to a computation of static properties albeit with a specific interpretation of the results. This is explored in the context of different application domains in Part IV of this thesis.

From a technical view point, *versatility* refers to the ability to support different techniques for the implementation of the desired functionality. While the structural part of a DFA specification is platform independent, the data-flow rules themselves have to be specified in an executable or interpretable language. Each language has its own benefits and drawbacks: OCL, for example, provides convenient methods for processing models but suffers from a limited functional range while Java is much more powerful but also tends to lead to more verbose specifications. An equivalent requirement can be formulated for the support of modeling standards. While MOF is the most commonly used modeling framework, competing techniques may already be in use. Therefore, the choice of the modeling standard as well as the implementation language should be up to the user. However, many underlying properties such as the notion of classes, associations or generalizations as basic constituents are shared by almost all modeling standards (e.g. KM3 [JBT06]). Defining the analysis methodology with respect to these properties therefore allows a simple adaption to other modeling frameworks.

6. **Performance and Scalability**
An important requirement for the practical applicability of any analysis technique is that it performs reasonably well for typical day-to-day applications. In this context, reasonably well means that the performance - i.e. the time and memory requirements - have to match the usage patterns in the common application scenarios. A live validation of user input has other time and memory constraints than an exhaustive analysis that can be executed over night as a batch job. In the ideal case, the technique should be scalable, depending only on the size of the input models. As an example for a concrete performance requirement, it could be determined that models in a certain domain typically contain less than a hundred elements in which case the live analysis should be

[16]The two application scenarios can be mapped to the typical application of attribute grammars and data-flow analysis in compiler construction: AGs are mostly used for validating static semantics while DFA extracts information that can be used to optimize the generated code.

completed in under a second. A practical evaluation of these properties in the context of different usage scenarios can be found in Chapter 10.

Challenges

The abstract goals must be aligned with the challenges that result from the differences between the technological spaces of compiler construction and modeling:

1. **Graph Structure of Models**
   As discussed in Section 4.1.4, models and context-free languages use different underlying structures for the internal (abstract) and external (concrete) representation of their language definitions and expressions. Programs written in a programming language, once processed by a parser, conform to syntax trees in which the edges denote the application of derivation rules. Models, on the other hand, possess the internal (and often also the external) structure of a graph with varying semantics concerning the interpretation of edges and nodes. This complicates mappings between both technological spaces as elements have to be processed based on their type. Additionally - in contrast to syntax trees and control-flow graphs - model graphs do not provide unique paths along which information can be routed throughout the model.

2. **Non-intrusive Design**
   Section 4.1.4 discussed how the OCL relates to the MOF language framework. The concepts that guide the extension of MOF with analysis language artifacts are also relevant for the description of a technique for the specification of flow analyses for existing modeling languages: Just like attribute grammars, OCL uses the existing MOF facilities for the definition of its own syntax and, at the same time, extends the language framework with a method that enables the annotation of derived languages with static constraints.

   This approach has multiple advantages: On the one hand, existing MDE facilities can be used as a basis for the specification process, e.g. to define a metamodel for OCL expressions. On the other hand, the tight integration with the target domain ensures a high level of compatibility during the application of the technique. The same principles should therefore form the basis for the definition of the DFA method. However, it is also important that compatibility with existing standards is preserved. Therefore, an OCL-like model-based extension mechanism should be employed that requires neither a modification of the MOF itself nor of any derived standard or tool.

3. **Fixed-point Semantics for the Modeling Domain**
   Traditional methods for computing fixed-point analyses of control-flow graphs are optimized for the requirements of the compiler construction domain. Consequently, as a prerequisite for the adaption to the modeling environment, the aspects that are specific to this area must be identified and addressed accordingly. At the same time, new requirements that arise from the transfer to the modeling domain must also be taken into account.

This pertains to several inherent properties of the flow analysis approach:

- The calculation of flow analysis results for models cannot rely on an automated propagation of (intermediate) results along predefined control-flow paths because of the aforementioned graph structure of models.
- Fixed-point computation traditionally imposes a set of constraints on value domains. Usually, it is required that the DFA results represent a finite, partial order of elements. This property guarantees that the calculation process always terminates in a unique fixed-point. As will be seen in the case studies in Chapter 10, not all analysis specifications which yield valuable results in the modeling domain are able to satisfy these prerequisites. For example, it may not always be possible to conveniently define a neutral element for confluence operators. Analyses may also employ complex datatypes for which it is difficult to devise a partial ordering. Nevertheless, through carefully constructed specifications and by incorporating the ability to handle these cases, it is possible to broaden the application areas of flow-based model analysis.
- For performance reasons, in traditional data-flow analysis, values are often represented as bitvectors. Support for complex datatypes which cannot be encoded in this way also affects the performance of the fixed-point computation. It is therefore necessary to provide efficient methods for calculating the DFA equation systems and to evaluate their performance in the context of realistic use cases.

4. **Flexibility on a Technological Level**
While a data-flow analysis itself is a declarative specification, the data-flow equations have to be written in an executable language. To allow for maximal flexibility, it should be possible to employ different languages for this task. This requires to define an interface that connects the definition of an analysis with the implementations of the data-flow rules:

   - On the one hand, this requirement influences the design of the analysis specification language which has to separate the declarative analysis definitions from the executable code. To provide support for alternative modeling frameworks besides MOF, this design should be based on a minimal set of core concepts shared by all modeling standards.
   - The interface must also enable the fixed-point evaluation algorithm to locate and invoke the rules and to process the respective results in a generalized, i.e. technology independent, way.
   - Finally, a methodology must be devised that allows data-flow rules - which may be written in an arbitrary language - to communicate with the DFA solver in a standardized fashion.

Although these points are mainly relevant to the actual implementation of the DFA approach in a software tool, they also have implications on the basic design of the approach with respect to the analysis specification language and the evaluation algorithm.

4.2.2. Conceptual Design and Language Artifacts

Based on the goals and challenges described in the last section, we are now able to describe the principal design of the approach both on a conceptual and on a technical level.

Attributed Models

In Section 4.1.4 and Section 4.1.5, we studied the extension of context-free grammars with semantic attributes. Through instantiation of the declarative specifications, context-sensitive information can be derived from syntax trees on *M1*. From the study of the conceptual similarities between modeling and formal languages, we can draw the conclusion that the method employed for annotating a grammar's constituents with analysis specifications is also a viable approach for use in the modeling space. The alignment of attribute grammars with the MOF modeling language framework therefore represents the basis for both the motivation and the actual implementation of this approach.

In Figure 4.10, we aligned attribute grammars with a - at the time - hypothetical method that supports the specification of flow analysis through the attribution of modeling languages. This concept comprises an extension of the *M3* language definition layer to support the annotation of language elements on *M2*. By using these constructs, (semantic) data-flow attributes can be assigned to classes in the target metamodel. The instantiation of these analysis specifications for *M1* models then results in attributed models in which attribute instances are attached to the respective model elements. The attributed model forms the input for a fixed-point evaluation that finally yields the DFA result values.

To realize the goal of *staying in the modeling domain* and to address the challenge of implementing *fixed-point semantics for the modeling domain*, we have to consider the characteristic properties of the target domain. For attributed models, this concerns - amongst other things - the generalization relationships between classes in a metamodel. Inheritance semantics demand that class attributes and operations defined in the context of a class are also made available at subclasses. To respect the practices in the modeling domain, this notion has to apply likewise to DFA concepts. In a sense, the notion of annotating semantic attributes at metamodel classes has a lot in common with MOF/UML class attributes. Therefore, we adapt the semantics of traditional class attributes for data-flow attributes: If a data-flow attribute is defined at a superclass, it is automatically inherited to specializations of this class. Additionally, just like class attributes can be redefined at subclasses, DFA attributes at a subclass override declarations of the same attribute at a superclass. The issue of instantiation semantics for attributes will be discussed subsequently in more detail.

From these observations, we can draw the conclusion that the attribution technique is a viable approach for the specification of flow analyses. This decision will now be further substantiated through a discussion of the benefits of this method:

Declarative Specification

The data-flow analysis method relies on the derivation of equation systems from language expressions which are subsequently subjected to fixed-point computations. Since the actual evaluation strategy is encoded in the fixed-point algorithm, the specifications themselves can be given in a declarative fashion which provides all necessary prerequisites for executing a flow analysis while at the same time eliminating the need for encoding specific evaluation semantics. As a consequence, the attribution concept is a good choice for the realization of a DFA methodology, since it inherently provides support for declarative specifications and is also well-suited for the usage in multi-layered language frameworks. Because of the declarative nature of attributions, the processes of analysis definition, instantiation and computation are also independent of the implementation languages used to encode the semantic rules. This fulfils the requirement of *versatility* since arbitrary (programming) languages such as Java or OCL can be used to specify the execution semantics of these rules.

The attribution method also provides support for different usage scenarios: An analysis defined by an attribution can be used to extract information from a model as well as for validating the syntactical structure of language expressions which also highlights the *versatility* of this approach.

The process of attributing metamodels can be simplified by providing a compact and concise analysis specification language, thereby supporting *ease of use*. To achieve the goal of *staying in the modeling domain*, we can employ the MOF framework to define this language in a similar way the syntax of attribute grammars is defined using EBNF. With respect to the definition of model analyses, existing MDE facilities such as editor generators can be used to complement this method with a matching tool chain.

Non-intrusive Design

A major advantage of employing a technique in the notion of attribute grammars for the definition of flow analyses is the fact that this approach allows a *non-intrusive* extension of existing languages with one or more analysis specifications. The annotation of semantic attributes requires only a unidirectional link connecting their specifications to elements in the target metamodel. Analysis artifacts can therefore be easily separated from the target language itself. As a result, the definition of an attribution specification language integrated with MOF's *M3* layer does not require any changes to the language framework. Consequently, this also applies to the artifacts on *M2*. In summary, to support the definition and assignment of attributes, no changes have to be made to existing standards and tools. This approach is therefore in line with the goal of providing *compatibility*.

Model-based Specification

As has been stated, the application of the attribution technique to the modeling domain is motivated by the conceptual ties between the two technological spaces. We described OCL as an example for a technique that both extends the capabilities of the modeling domain and at the same time makes use of the existing facilities to define its own syntax. By applying the same principles to a model-based attribution technology, we can achieve the goal of *staying in the modeling domain*. Because the specifications themselves are based on MDE techniques and standards, the analyses can be defined using exactly the same modeling facilities on which the target modeling language is based and thus no mapping has to be provided between different formalisms.

On a conceptual level, the integration with the modeling space requires to provide support for the relevant features found in this domain. As has already been mentioned, in the context of declarative specifications that extend classes in the metamodel, this means that the defined generalization hierarchy must also be respected by the attribute extensions. For data-flow attributes, we can implement this behavior through the adoption of MOF generalization semantics. Using the attribution formalism as a basis for flow analysis specification also has another advantage: Just like classes can contain multiple class attributes, it is possible to assign more than one DFA attribute to a class. This is an important feature for incrementally building more complex analyses by reusing the results of one or more DFA attributes in the computation of the result of other attributes.

Generality

The declarative nature of attributions also abstracts from the model's inherent graph structure. The data-flow paths used for the propagation of information from one model element to another[17] are not part of the attribution specification (see below). Therefore, the definition of an attribution solely relies on two metamodel concepts, classes and generalizations. The classes represent the context for the annotation of semantic attributes while generalization relationships influence the availability of attributes at subclasses. This satisfies the *versatility* requirement with respect to the applicability to different modeling paradigms because the approach can be implemented for all modeling frameworks that include support for these two concepts.

Based on this observation, the process of computing DFA results can be divided into two phases: First, data-flow attributes are instantiated for a given model taking into account the generalization hierarchy in the metamodel. In the second phase, the instantiated attribution, i.e. the attributed model, is subjected to a fixed-point evaluation that yields the result values for the attribute instances.

[17]More specifically, information is propagated from an attribute instance annotated at a model element to another attribute instance.

Language Artifacts

One step in the process of implementing the attribution approach consists of the construction of a suitable specification language. This language has to enable the annotation of semantic attributes at classes in a target metamodel. These attributes can then be instantiated for arbitrary models.

We concluded that this language should itself be based upon MDE principles, i.e. its specification has to adhere to the practices of the MOF language framework. Consequently, we have to define an abstract syntax in the form of a metamodel. To allow the assignment of semantic attributes to classes, the metamodel of the analysis specification language has to establish a connection between attribution concepts and MOF's *M3* layer. The abstract language definition can subsequently be complemented with a concrete, human-readable syntax.

In the previous paragraphs, we motived our choice of using the attribute grammar formalism for the specification of data-flow analyses. We will now derive the necessary components that comprise a declarative specification of semantic attributes from the structural composition of attribute grammars. The adaption of these constructs for our use case has to follow the notion of assigning attributes to context-free grammars: Starting with a set of attributes, occurrences of these attributes can be attached to metamodel classes. An attribute occurrence therefore connects a specific attribute and a data-flow rule used to compute a result value to a target class.

This structure maps the abstract syntax of an attribution to a textual representation and vice versa. Since the interface between an attribution and the target metamodel consists only of the annotation relationships between occurrences and their target classes, we can represent attributions in a similar way to OCL constraints: Attribution specifications can be written in a textual format that references the target metamodel (just like OCL constraints define a context class). Alternatively, attribute occurrences could be represented as objects that are connected to classes inside the metamodel's diagram.

In the following, we list the concepts that constitute the abstract syntax of the analysis specification language. These elements realize an adaption of the attribute grammar formalism to the requirements of the modeling domain.

Datatype
: Traditionally, results of data-flow analyses are sets containing elements from a specific value domain. As mentioned in the challenge of providing *fixed-point semantics for the modeling domain*, it can be beneficial to lift this limitation to allow for user-defined result types. This is achieved through the Datatype concept that defines custom types for the contents of attribute results. While the Datatype itself can be arbitrary, its use however has to be consistent across all occurrences of a specific attribute. It is important to note that the actual interpretation of values corresponding to a type depends on the language that is used to implement the Data-flow Rules.

Data-flow Rule
: As stated in the challenge of *flexibility on a technological level*, it should be

possible to specify the execution semantics of data-flow equations using arbitrary programming languages. A Data-flow Rule (or semantic rule) in the context of the attribution language therefore has to encode both the actual implementation of the rule as well as information that is required to invoke it.

In traditional data-flow analysis, result variables are initialized with a value that represents a neutral element with respect to DFA operations (such as $\emptyset$ for the union operator), while the equations are used to calculate the instances' iteration values during the fixed-point computation. However, with the introduction of custom Datatypes, the nature of neutral elements depends on the actual type of the result values. To address this issue, we state that Data-flow Rules may not only be used to specify iteration rules but also for initialization purposes, i.e. the assignment of initial values to attributes.

Attribute Definition

Attribute Definitions constitute the type definitions of an attribution. An attribute is characterized by a unique name, a Datatype and a Data-flow Rule which yields the initialization values for all instances of this attribute.

In the design goal of *versatility*, we discussed two different application scenarios: Attribute Definitions (and their derived instances) can either be used to approximate dynamic properties of models or to validate their well-formedness. We therefore distinguish between Assignment attributes which can compute result values of arbitrary Datatypes and Constraint attributes that evaluate to the boolean values of true or false, indicating whether the associated structural restriction has been met.

Attribute Occurrence

In the attribute grammar formalism, a single attribute may have multiple occurrences in different production rules. Correspondingly, we introduce an Attribute Occurrence concept that indicates the presence of an Attribute Definition at a specific metamodel class. This is referred to as an *occurrence* of this attribute at the respective class. Each definition can therefore possess an arbitrary number of occurrences at different classes.

In attribute grammars, each occurrence is supplied with a semantic rule that describes how to calculate its value depending on the respective context. The same concept also applies in our scenario: While all occurrences of an attribute are initialized with the same value, the computation of their iteration values may differ. Therefore, an Attribute Occurrence also has to declare a Data-flow Rule for the calculation of its fixed-point iteration results.

Attribute Instance

The concepts listed above constitute the attribute specification language. To instantiate an attribution for a *M1* model, each occurrence of an Attribute Definition has to be instantiated for all elements that conform to the type of the occurrence's target class. This process results in a set of Attribute Instances which are connected to elements in the target model and carry a value of the attribute's Datatype.

As has been described above, the instantiation process has to take into account the generalization relationships between classes. Inheritance semantics demand that an Attribute Occurrence attached to a target class is also implicitly defined for subtypes. This leads to another problem that must be addressed: When specifying the instantiation mechanism, it is also necessary to consider the case of redefinition, because an occurrence of an attribute at a subtype may override the definition of an occurrence of the same attribute at a superclass.

While it is not essential to define an abstract syntax for Attribute Instances, doing so has several benefits: A model-based representation of Attribute Instances is able to provide a clear definition of their properties and their relationships with other modeling artifacts such as the target model and the analysis specification. This approach therefore supports the goal of *staying in the modeling domain*. For example, one can directly navigate from an Attribute Instance element in the attributed model to its defining Attribute Occurrence using default MDE methods.

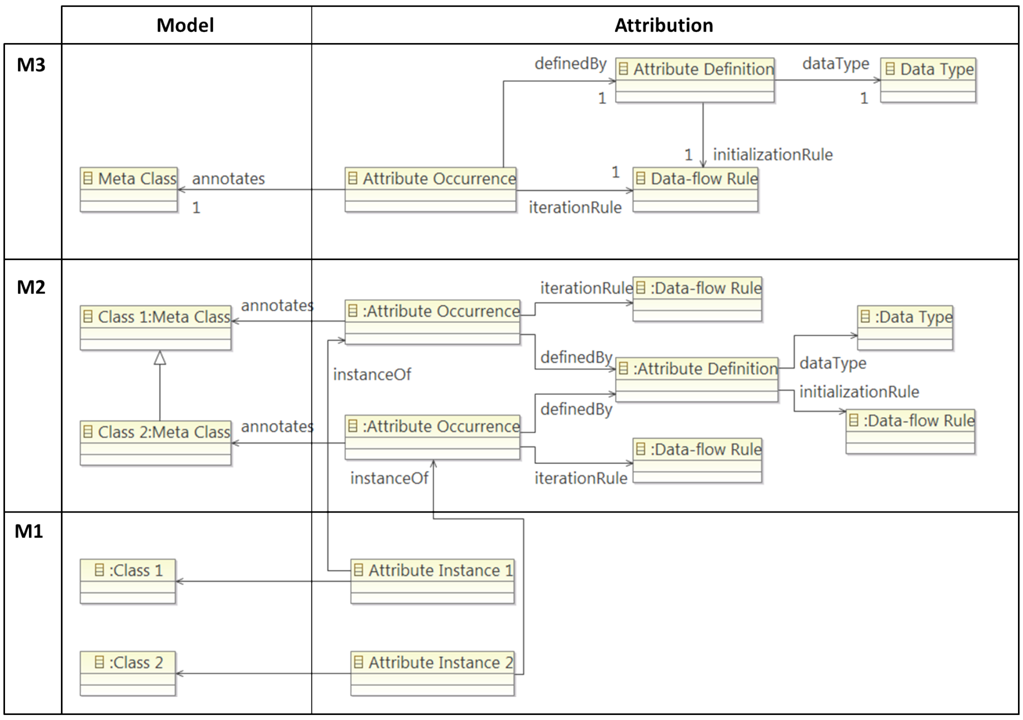


Figure 4.11.: Extending the MOF language framework to support attributed models.

Effectively, the presented concepts amount to a domain-specific language for the specification of data-flow based model analyses in the notion of attribute grammars. The definition and usage of this language is illustrated in Figure 4.11. In this schematic, the MOF language framework is extended with the ability to define attributions for metamodels which in turn leads to attributed models on *M1*. These can then be subjected to a fixed-point evaluation.

On the top level, the attribution specification language itself is defined and a (unidirectional) link to the Class concept in MOF is established. Attributions can now be defined for metamodels on *M2*. In this example, two occurrences of the same Attribute Definition are attached to Class 1 and Class 2 in the target metamodel. Each occurrence uses a separate Data-flow Rule to calculate DFA iteration values while the definition uses a third rule that provides the initialization values for the fixed-point computation. Because of the generalization relationship between the two classes, the occurrence annotated at Class 1 is overwritten at Class 2. Instantiating this analysis specification for the displayed model consequently yields two Attribute Instances on the *M1* layer. Note that because of the redefinition at Class 2, the second instance's iteration value will be computed with a different Data-flow Rule than the first one's.

Just as is the case for any model-based DSL, the development of the analysis specification language requires the definition of the following artifacts:

Abstract syntax
The abstract syntax defines the structure of model-based attributions. In the modeling domain, the abstract syntax is specified as a metamodel. This metamodel has to encode the concepts listed above along with their properties and relationships. Furthermore, it has to define a connection point to MOF, i.e. implement the possibility to annotate classes in a given target metamodel with Attribute Occurrences.

Concrete syntax
The abstract syntax has to be complemented by a concrete representation. Because of the declarative nature of the attribution method and since its characteristics were derived from the technique of attribute grammars, it can be assumed that the preferred method for representing attribution specifications is a textual format.

Static semantics
As mentioned in Section 4.1.5, not all properties that constitute the well-formedness of a model can be encoded in the abstract syntax. Naturally, this restriction will also apply to the metamodel that describes the abstract syntax of the analysis specification language. Therefore, it is necessary to state additional constraints that will ensure that an analysis is valid, i.e. it constitutes a valid specification that can be evaluated by a fixed-point algorithm.

Dynamic semantics
In the context of the proposed approach, the dynamic semantics of attributions have to reflect how an analysis specification can be instantiated and evaluated for a given model. The challenges that arise from the adaption of the DFA method to the modeling approach will be discussed in the next paragraphs.

As [Bra09] states: *"The development of domain specific modeling formalisms involves more than just the development of metamodels or profiles. It also involves the* *development of tooling, methodology and even validation of the modeling formalism via empirical research".* These issues will be addressed in the following chapters: While Chapter 5 focuses on the theoretical aspects of the approach, Chapter 6 provides detailed technical specifications of the presented concepts based on MDE technologies. Chapters 7 and 8 demonstrate how the proposed approach can be implemented by devising a suitable architectural design and describing a prototypical tooling environment.

Information Propagation

In contrast to control-flow graphs, models do not possess a corresponding backbone structure that provides an inherent flow direction along which information can be automatically propagated (forward vs. backward flow analysis). Neither do models impose a hierarchical order on the elements of the language instance as is the case with attributed syntax trees (inherited vs. synthesized attributes).

A possible solution to this problem would be to explicitly define in the attribution specification the routes along which information should be propagated. These route configurations would need to describe how information flows from one element to the other, or in other words, which values are provided as input for the computation of a specific attribute instance. For example, it could be stated that results calculated at model elements of class type C_A should always be propagated to elements of type C_B along the association $A_{C_A \rightarrow C_B}$ connecting those elements. The fixed-point solver could then interpret these specifications before the actual computation takes place and build the corresponding dependency graph for the attributed model.

However, the need for a separate definition of information propagation paths would violate the goal of *ease of use*. Moreover, it also has a negative impact on the *performance and scalability* requirement. This reason for this is that specifying route configurations on the meta level can lead to an over-approximation of the dependencies between attribute instances which in turn has a negative effect on the performance of the fixed-point computation since more iterations may be necessary until a stable fixed-point is reached. This can be illustrated through an example: We assume that information should only be propagated from a model element of type C_A to an element of type C_B if a specific condition is met that cannot be verified by the solver[18]. This restriction therefore cannot be included in a route configuration on the meta level. The definition of an output dependency $C_A \longmapsto C_B$ would consequently establish dependency relationships between all elements of types C_A and C_B while, in reality, only a subset of these dependencies is relevant to the computation.

To solve this problem, we propose a different approach that does not require explicit specifications of information propagation paths but instead embeds the dependencies inside the data-flow rules themselves. This can be achieved through a dedicated operator that is made available in the rule implementation language's syntax to access instances of data-flow attributes and retrieve their values. Using this method, it is no longer necessary to explicitly state the dependency $C_A \longmapsto C_B$.

[18] For example, if the condition itself is computed by a data-flow analysis and thus is not available at the start of the result computation.

Rather, the implementation of a data-flow rule of an attribute assigned to class C_B actively requests the value of a specific instance of type C_A as input and thereby establishes a dependency relationship between exactly these two *M1* objects. However, this approach introduces a new challenge: Because these requests occur during the fixed-point computation, the dependency graph cannot be constructed ahead of time but rather has to be built dynamically during the execution of the rules by monitoring and recording the accesses made by the DFA rules. This requires an adaption of the traditional solving algorithms to provide support for the dynamic discovery of new dependency relationships during the solving phase and the incorporation of these dynamic dependencies in subsequent fixed-point iterations.

However, as another positive side-effect, this approach also eliminates the problem of different edge semantics. The reason for this is that the dynamically discovered dependencies result in a dependency graph that superimposes the model graph. Strictly speaking, information is not propagated along model edges but rather along the edges of the dependency graph. Since the structure of this graph may differ from the layout of the model, this property further simplifies the definition of the information flow as information can be routed along arbitrary paths. It is, for example, possible to envision a use case where information should be propagated from an ActivityNode in an UML Activity Diagram to its successor node. Instead of routing the data from $Activity_1$ to the outgoing ActivityEdge $Edge_{Activity_1 \rightarrow Activity_2}$ from where it is again forwarded to $Activity_2$, we can propagate the DFA value directly from $Activity_1$ to $Activity_2$, skipping the intermediate edge. This approach therefore also provides *flexibility on a technological level*.

Instantiation of Flow Analyses for Models

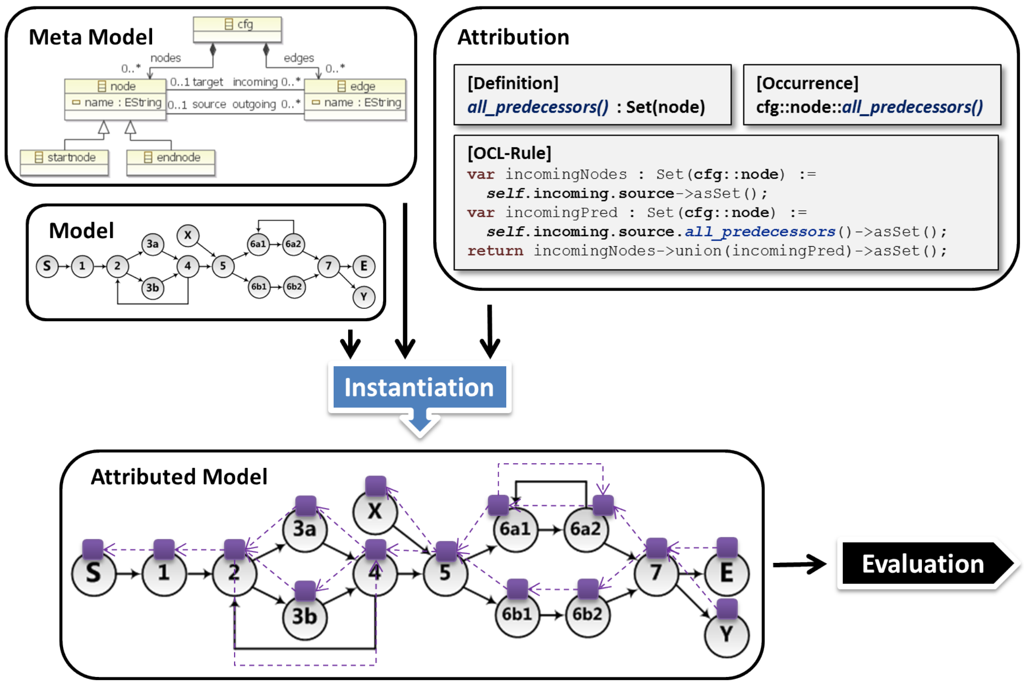
As a prerequisite for the execution of an analysis, the analysis specification must first be instantiated for the target model. In essence, for each occurrence of an attribute assigned to a class, a corresponding attribute instance has to be created at model elements of this type. In the absence of generalization relationships, this process is straightforward. If, however, a class inherits from another class, this property has to be reflected in the instantiation process.

In [SB13], we first described the properties of attributed metamodels:

> The instantiation semantics for attributes follows the EMOF semantics for the instantiation of metamodel classes: An attribution AT(MM, AT_{DEF}, AT_{RULE}, AT_{OCC}, AT_{DT}, AT_{TYPE}, AT_{ANN}) extends a metamodel MM(MM_{CL}, MM_{GEN}) given by the set of classes MM_{CL} and their generalization relationships MM_{GEN} indicating inheritance of structural and behavioral features in accordance to EMOF semantics. The attribution consists of attribute definitions AT_{DEF}, each possessing a datatype (AT_{DT}) and an initialization rule (AT_{RULE}) assigned by the relation AT_{TYPE}. Furthermore, the annotation relation AT_{ANN} ties each occurrence in AT_{OCC} to a class $c \in \text{MM}_{CL}$ and an iteration rule in AT_{RULE}.

We then went on to outline the instantiation process itself:

An instantiation INST(AT, M, INST_{AT}, INST_{LINK}) contains attribute instances INST_{AT} for an attribution AT and a model M $\lhd$ MM with objects M_{OBJ} and a relation M_{TYPEOF} denoting their class type. For each *obj* $\in$ M_{OBJ}, an attribute instance $i \in$ INST exists *iff* there are ≥ 1 occurrences *occ* $\in$ AT_{OCC} for the class type of *obj* or its super-types. To realize overwriting at subtypes the most specialized type is used. This can be implemented by starting at a model object's concrete type and traversing the generalization hierarchy upwards. For the first occurrence of each distinct attribute definition which is encountered an instance is created. Multiple inheritance is only supported if generalization relations are diamond-shaped and a unique occurrence candidate can be identified.



[Definition]
all_predecessors() : Set(node)

[Occurrence]
cfg::node::*all_predecessors()*

[OCL-Rule]
```
var incomingNodes : Set(cfg::node) :=
  self.incoming.source->asSet();
var incomingPred : Set(cfg::node) :=
  self.incoming.source.all_predecessors()->asSet();
return incomingNodes->union(incomingPred)->asSet();
```

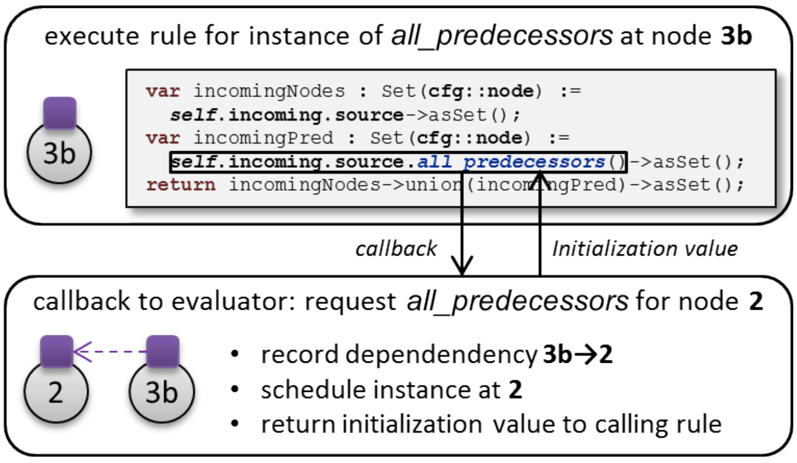
Figure 4.12.: Instantiation of an attributed metamodel.

The instantiation process is exemplified in Figure 4.12. The artifacts given as input are the metamodel, an attribution of that metamodel and the model for which this analysis should be executed. In this example, the metamodel describes a simple control-flow graph while the attribute allPredecessors computes the set of transitive predecessors for each node using a DFA rule written in imperative OCL. Based on this input, the attribution can now be instantiated for arbitrary models. In this case, attribute instances are created for and assigned to each element of type node. While the illustration indicates the existence of data-flow dependencies with dashed lines, it is important to note that these dependencies are not yet known at this point since they are encoded in the OCL rule and will only become apparent when this rule is invoked during the fixed-point computation.

Fixed-Point Evaluation with Dependency Discovery

The need for a customized version of a fixed-point evaluation algorithm stems from the fact that associations in a metamodel do not necessarily provide meaningful paths for the propagation of data-flow information. The starting point for the implementation of an efficient and scalable evaluation method is the worklist algorithm traditionally used in compiler construction. This algorithm must be adapted to provide support for the challenges resulting from the graph structure of models, i.e. the dynamic discovery of the dependencies implicitly encoded in the rules. More specifically, the DFA solver must be extended with the ability to record the input/output relationships between attribute instances and incrementally build the attribute dependency graph. Based on this graph, the solver can then derive an optimized scheduling for the execution of the iteration rules. Additionally, knowledge about the dependencies can also be used to parallelize the execution of the data-flow equations in unrelated branches of the graph. Finally, the algorithm should also support on-demand analysis: Supplying a subset of attribute instances that are of interest, the solver should be able to automatically expand this set to include depending instances.

The cornerstone of this algorithm is a callback mechanism that provides DFA rules (executed in the context of a specific instance) with the ability to request other attribute instance values as input. This is in contrast to the traditional approach where inputs are supplied automatically based on the structure of the underlying flow graph. By relaying the request for an input back to the solver, it is able to record the hitherto unknown dependency between the requesting and the requested instance. Repeating this process for all relevant instances yields the overall dependency graph that describes the data-flow paths. Since the starting set of instances can be chosen arbitrarily, support for lazy evaluation strategies is inherently provided. Obviously, the successful implementation of this approach relies on the definition of language interfaces that allow implementations of data-flow rules to actively request the values of other attribute instances.


execute rule for instance of *all_predecessors* at node **3b**
3b
callback
Initialization value
callback to evaluator: request *all_predecessors* for node **2**
2
3b
record dependency **3b→2**
schedule instance at **2**
return initialization value to calling rule

```
var incomingNodes : Set(cfg::node) :=
  self.incoming.source->asSet();
var incomingPred : Set(cfg::node) :=
  self.incoming.source.all_predecessors()->asSet();
return incomingNodes->union(incomingPred)->asSet();
```

Figure 4.13.: Recording implicit dependencies through a callback to the evaluator.

Figure 4.13 demonstrates the callback mechanism based on the example shown in Figure 4.12. In this case, we assume the solver is currently interested in evaluating the instance of the attribute allPredecessors which is located at node ③b. For this purpose, the solver invokes the iteration rule assigned to the instance's corresponding attribute occurrence. In the fourth line of the OCL code, the navigation statement self.incoming.source collects the direct predecessors of the current node, skipping the intermediate edges. For ③b, this results in the set {②}. Then, the OCL interpreter invokes the attribute access operation $allPredecessors()$[19] on the elements in this set. This method triggers a callback which informs the solver that the instance of allPredecessors at node ③b has requested the value of the instance of the same attribute at ② as input. The solver can now record the discovered dependency and schedule the instance at ② for a subsequent evaluation. In the meantime, the rule of the requesting instance is supplied with the defined DFA initialization value and is able to continue with its execution.

[19]The operation $allPredecessors()$ is a black-box method that has been injected into the OCL environment for elements of the type node. The information required to automate this process can be derived from the attribution. This method is detailed in Section 6.4.

5. Formal Semantics of Flow-based Model Analyses

In the last chapter, we discussed conceptual and technical properties of the target domain, the MDE ecosystem, and the resulting implications on the transfer of the DFA method. Based on these considerations, we will now develop a formal definition of the semantics of flow analysis annotations at metamodels. This is an essential requirement as only a sound formal model is able to provide a solid foundation for an in-depth understanding of this technique. The importance of this endeavor is evident from the effort which has been directed at the formalization of the Object Constraint Language which - in some sense - can be viewed as a precursor of the flow-based analysis approach. The authors of [RG98] state that the OCL is *"mainly defined in a semi-formal way by using English text descriptions, a context-free grammar specifying the concrete syntax [...] and many examples illustrating the intended meaning of expressions"* and argue that *"[w]hile this style of presentation is perfectly well-suited to introduce and demonstrate the concepts of OCL, [...] a thorough understanding of OCL semantics requires a formal definition"*. In [CK04a], it is mentioned that since the *"navigational expression language forms the basic building blocks from which the requirements on system states and executions may be constructed [...] the practical use of the OCL thus depends not to the least degree on a clear understanding of the basic OCL expressions"*. It is further emphasized that *"[a] clear semantic foundation of the OCL in general represents a sine qua non if we want to put OCL on equal footing with other well-established model specification languages"*.

The task of providing a concise description of the proposed method requires the choice of a suitable formalism for this purpose. Multiple approaches exist in related research fields which are applicable in different contexts and thus possess individual strengths and weaknesses. In many cases, these techniques are based on formal logic or set theory. Methods which have already been applied in the modeling domain include [BW02; CK01; Var02; MB06]. For obvious reasons, it is preferable to select a suitable formalization methodology from the repertory of methods which have already been successfully put to use for similar purposes. For the formalization of the flow-based methodology, we have chosen to base our specifications on the method which has been used by Cengarle & Knapp in [CK01; CK04a] to define big-step operational semantics for OCL expressions. This choice was motivated by the assumption that the conceptual similarities between OCL and the DFA method would simplify the adaption of the original approach: Analysis specifications, whether they are written in OCL or encode a DFA, represent an extension of an existing metamodel. As noted by the authors of [RG98], the *"primary purpose* *of OCL is to augment a model with additional information that often (if at all) cannot be expressed appropriately in UML"* and that the *"intended effect of a constraint specification is a restriction on possible system states and transitions with respect to a given model"*. Therefore, it is necessary to provide a suitable representation of the relevant properties of (meta)models for the subsequent specification of the evaluation semantics. Furthermore, an *"OCL expression is declarative in the sense that an expression says what constraint has to be maintained, not how this is accomplished. Therefore, specification of constraints is done on a conceptual level, where implementation aspects are mainly irrelevant"*. These properties also hold for the flow analysis technique. The reuse of an approach which has already been successfully employed to specify the semantics of OCL expressions therefore can be expected to streamline the formalization of our own method.

This chapter is divided into two parts: In Section 5.1, we present a formal abstraction of metamodels and models along with accompanying evaluation semantics for expressions. These definitions are then extended in Section 5.2 to include support for data-flow attributes both on the meta and the instance layer. Correspondingly, the evaluation semantics are enhanced with support for the computation of interdependent attribute instances. This includes an algorithm for the derivation of fixed-point results. Finally, we employ the formal specification to assert a number of essential claims.

5.1. Formal Semantics for Models

In this section we introduce a formalization of elementary modeling concepts on the metamodel and model layer along with corresponding evaluation semantics for expressions. These descriptions form the foundation for the subsequent formal specification of data-flow annotations at metamodels and their evaluation in Section 5.2. The formalization is inspired by the methodology employed by [CK01; CK04a] for the definition of big-step operational semantics for the Object Constraint Language. Section 5.1.1 defines the basic syntactic elements of the target domain, namely class and object descriptors which represent the model and metamodel artifacts and a syntax for expression statements. In Section 5.1.2 we then present evaluation semantics which are able to interpret expressions in the context of model elements. As mentioned, these concepts form the basis for the attribute-based extensions in the next section which describe the definition and evaluation of data-flow analyses.

5.1.1. Class and Object Descriptors

The descriptions in this section, which we will incrementally develop, are based on a formalization which encodes the relevant syntactical aspects of metamodels, models and expressions.

Definition 5.1.1 (*Class descriptor*)

The signature of a *class descriptor* $\mathcal{C}$ is given as

$$(C, Q), \text{ where}$$

i. C is a set of classes and types

ii. Q is a set of query signatures in the form

$$q : c_{ctx} \times c_1 \times \ldots \times c_n \to c_{ret}, \text{ with}$$

- query context $c_{ctx} \in C$
- input parameters $c_1, \ldots, c_n \in C$
- return type $c_{ret} \in C$

Definition 5.1.1 introduces the concept of *class descriptors*. A class descriptor $\mathcal{C}$ conforms to a simplified formal representation of the respective target metamodel. More specifically, the tuple (C, Q) describes the classes and types contained in the metamodel as well as concepts defined by the underlying modeling framework (including e.g. primitive data types). Furthermore, it specifies a set of query signatures which denote functions that operate on the available types. A query is always defined in the context of a specific class c_{ctx}, possesses a set of input parameters $c_1, \ldots, c_n$ and returns an output element c_{ret}. For clarity reasons, we omit features such as a sophisticated typing system, generalization hierarchies and associations between classes. It should be noted that this does not result in a loss of generality as the proposed flow analysis methodology relies on a very basic set of fundamental properties. Semantics for omitted features can be easily inferred from the listed specifications. The exhaustive definitions provided by [CK04a] can be used as a starting point.

Definition 5.1.2 (*Object descriptor*)

An *object descriptor* $\mathcal{I}$ over a class descriptor (C, Q) is given as

$$(C^{\mathcal{I}}, Q^{\mathcal{I}}), \text{ where}$$

i. $c^{\mathcal{I}} \in C^{\mathcal{I}}$ is a set of objects for each class $c \in C$

ii. $q^{\mathcal{I}} \in Q^{\mathcal{I}}$ is a function for each query $q \in Q$ in the form

$$q^{\mathcal{I}} : c^{\mathcal{I}}_{ctx} \times c^{\mathcal{I}}_1 \times \ldots \times c^{\mathcal{I}}_n \to c^{\mathcal{I}}_{ret}$$

We further require that the interpretation includes non-termination ($\perp$):

$$\perp \in c^{\mathcal{I}} \text{ for all } c \in C$$

The concept of an *object descriptor* $\mathcal{I}$ is formalized in Definition 5.1.2. This structure represents an interpretation of a given class descriptor (C, Q) and thereby conforms to an instance of the respective language. An object descriptor $\mathcal{I}$ consists of interpretations of classes $c \in C$ and queries $q \in Q$. A specific interpretation of a class represents the set of objects in the target model which are of the respective type. Query interpretations represent functions which are interpreted in the context of an element of type c_{ctx} and map input parameters $c_1^{\mathcal{I}} \times \ldots \times c_n^{\mathcal{I}}$ to an output object or value in $c_{ret}^{\mathcal{I}}$. Since, in the case of cyclic dependencies between queries, the evaluation of an expression may result in non-termination, the element $\perp$ has to be included in the set of potential results.

Definition 5.1.3 (*Expression syntax*)

Expressions e are defined as

$$\begin{aligned} e ::= {} & \texttt{self} \\ & \mid e.q(e_1, \ldots, e_n) \text{ with } q \in Q \\ & \mid \texttt{if } e \texttt{ then } e_1 \texttt{ else } e_2 \texttt{ endif} \end{aligned}$$

Definition 5.1.3 provides a syntax for simple expressions. Again, we restrict ourselves to the examination of a reduced set of features to enable a short and concise formalization of the evaluation semantics. More specifically, we study only statements which include the `self` reference (which denotes the context object) and queries q (as defined in the class and object descriptors) which possess an arbitrary number of input parameters. To exemplify the application of this methodology for common expression types which are an essential part of languages such as OCL, we additionally include the conditional if ... then ... else operator. Using this example as a starting point, it is possible to extend the specifications to provide support for additional functionality such as iteration or navigation statements.

5.1.2. Expression Semantics

Based on the definition of the syntactical properties of (meta)models and expressions, it is now possible to specify the semantics of expression statements.

Definition 5.1.4 (*Evaluation semantics for expressions*)

$$(\texttt{self}^{\downarrow}) \quad \mathcal{I}, \gamma \vdash \texttt{self} \downarrow \gamma(\texttt{self})$$

$$(\mathrm{q}^{\downarrow}) \quad \frac{\mathcal{I}, \gamma \vdash e \downarrow v \qquad (\mathcal{I}, \gamma \vdash e_i \downarrow v_i)_{1 \leq i \leq n}}{\mathcal{I}, \gamma \vdash e.q(e_1, \ldots, e_n) \downarrow q^{\mathcal{I}}(v, v_1, \ldots, v_n)}$$

$$(\mathrm{cond}_1^{\downarrow}) \quad \frac{\mathcal{I}, \gamma \vdash e \downarrow \texttt{true} \qquad \mathcal{I}, \gamma \vdash e_1 \downarrow v_1}{\mathcal{I}, \gamma \vdash \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2\ \texttt{endif} \downarrow v_1}$$

$$(\mathrm{cond}_2^{\downarrow}) \quad \frac{\mathcal{I}, \gamma \vdash e \downarrow \texttt{false} \qquad \mathcal{I}, \gamma \vdash e_2 \downarrow v_2}{\mathcal{I}, \gamma \vdash \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2\ \texttt{endif} \downarrow v_2}$$

Definition 5.1.4 lists a set of inference rules which describe natural semantics for the derivation of judgements for expressions which conform to Definition 5.1.3. Statements in the form $\gamma \vdash \sigma \downarrow \eta$ denote that the evaluation of a statement σ in a system state γ results in the value η. We require that γ contains a binding for `self` and thus the context object can be retrieved via $\gamma(\texttt{self})$. The upper part of each rule identifies the hypothesis for the application of the rule while the lower section denotes the respective judgement. The evaluation of a (partial) expression relies on an environment which consists of an object descriptor $\mathcal{I}$ and a variable binding for `self`. Evaluation is non-strict for if ... then ... else.

The first rule, which does not have a prerequisite, states that `self` always evaluates to o for a variable binding $\texttt{self} \mapsto o$ in γ. The evaluation of a query q on an expression e requires the evaluation of both the context and the input parameters and applies the function $q^{\mathcal{I}}$ to determine the query's result value. The interpretation of if ... then ... else statements depends on the value derived for the condition e. If the condition evaluates to true, the overall result corresponds to the evaluation of the then branch. Otherwise, the value which is yielded by the else branch is returned.

5.2. Formal Semantics for Attributed Models

We will now extend the definitions from Section 5.1 to provide support for the declaration and evaluation of data-flow attributes. For this purpose, it is necessary to enhance the definitions of class and object descriptors as well as the syntax of expression statements with attributes. This is detailed in Section 5.2.1. In the next step, we adapt the evaluation semantics to ensure a correct handling of expressions in attributed models. Since there exist multiple strategies to address this problem, we chose to present two different approaches which can be used interchangeably: The semantics specified in Section 5.2.2 strictly limit the evaluation of attribute dependencies, thus preventing recursive calls. On the other hand, the definition in Section 5.2.3 allows multiple consecutive recursion steps, halting only if circular dependencies are encountered to prevent non-determination. Both specifications additionally rely on semantic functions which control the iterative evaluation to enable the computation of the desired fixed-point results. Finally, in Section 5.2.4, we state two claims which assert specific properties that hold for the proposed semantics and provide the necessary proofs.

5.2.1. Attributed Class and Object Descriptors

The following descriptions extend the definitions from Section 5.1.1 with support for data-flow attributes.

Definition 5.2.1 (*Attributed class descriptor*)

The signature of an *attributed class descriptor* $\mathcal{D}$ is given as

$$((C, Q), A), \text{ where}$$

i. (C, Q) is a class descriptor

ii. A is a set of attributes in the form

$$a : c_{ctx} \rightarrow c_{ret} = E_a$$

which assigns an expression E_a to each attribute $a \in A$.

Definition 5.2.1 enhances the signature of class descriptors (cf. Definition 5.1.1) with a set of data-flow attributes A which represent the attribution of an existing metamodel. The resulting structure is termed an *attributed class descriptor*. An attribute is defined in the context of a class c_{ctx} and produces an output of type $c_{ret} \in C$. The result for an attribute is computed by an expression E_a (cf. Definition 5.2.3) which in turn may rely on other attributes.

Definition 5.2.2 (*Attributed object descriptor*)

An *attributed object descriptor* $\mathcal{O}$ over an attributed class descriptor $((C, Q), A)$ is given as

$$(\mathcal{I}, \mathcal{A}), \text{ where}$$

i. $\mathcal{I}$ is an object descriptor

ii. $a^{\mathcal{A}} \in \mathcal{A}$ is an interpretation of the attribute $a : c_{ctx} \rightarrow c_{ret} = E_a \in A$ with

$$a^{\mathcal{A}} : c_{ctx}^{\mathcal{I}} \rightarrow c_{ret}^{\mathcal{I}}$$

Definition 5.2.2 extends conventional object descriptors (cf. Definition 5.1.2) to specify the concept of *attributed object descriptors*. This process mirrors the extension of class descriptors and delineates the structural properties of attributed models. For this purpose, a set $\mathcal{A}$ is included in the descriptor which consists of interpretations of the data-flow attributes $a \in A$. Consequently, $a^{\mathcal{A}}(o)$ retrieves the current value of an attribute of type a in the context of an object o. It should be noted that the state of $\mathcal{A}$ may change over time due to the iterative fixed-point evaluation process which repeatedly updates the values of attribute instances $a^{\mathcal{A}}(o)$.

Definition 5.2.3 (*Expression syntax (including attributes)*)

Expressions with attributes E are defined as

$$\begin{aligned} E ::= {} & \texttt{self} \\ \mid {} & E.q(E_1, \dots, E_n) \text{ with } q \in Q \\ \mid {} & E.a \text{ with } a \in A \\ \mid {} & \texttt{if } E \texttt{ then } E_1 \texttt{ else } E_2 \texttt{ endif} \end{aligned}$$

Definition 5.2.3 extends the expression syntax from Definition 5.1.3 with support for attributes. Similar to queries, occurrences of attributes $a \in A$ in an expression E denote properties which have to be evaluated the context of an object. However, as mentioned above, the evaluation of an attribute may lead to an update of the instantiated attribution $\mathcal{A}$. Furthermore, if an attribute depends on other attributes (resulting in circular dependency relationships) this situation does not result in non-termination (assuming the value domain and the operations conform to the requirements of the data-flow analysis approach as stated in Section 2.3.2).

5.2.2. Non-recursive Expression Semantics

In the following, we present inference rules for the evaluation of attributed object descriptors. This specification relies on the combination of two sets of rules to manage the potentially cyclic dependencies between attributes. In this case, dependencies between attributes are not regarded. Instead, the computation is controlled by an external algorithm which resembles the round-robin strategy for solving fixed-point equation systems.

Definition 5.2.4 (*Expression semantics: direct evaluation*)

(self$^{\downarrow}$) $\mathcal{A}, \gamma \vdash^{\mathcal{I}}_{d} \texttt{self} \downarrow \gamma(\texttt{self})$

(q$^{\downarrow}$) $$\frac{\mathcal{A}, \gamma \vdash^{\mathcal{I}}_{d} E \downarrow v \qquad (\mathcal{A}, \gamma \vdash^{\mathcal{I}}_{d} E_i \downarrow v_i)_{1 \leq i \leq n}}{\mathcal{A}, \gamma \vdash^{\mathcal{I}}_{d} E.q(E_1, \ldots, E_n) \downarrow q^{\mathcal{I}}(v, v_1, \ldots, v_n)}$$

(a$^{\downarrow}$) $$\frac{\mathcal{A}, \gamma \vdash^{\mathcal{I}}_{d} E \downarrow v}{\mathcal{A}, \gamma \vdash^{\mathcal{I}}_{d} E.a \downarrow a^{\mathcal{A}}(v)}$$

(cond$^{\downarrow}_1$) $$\frac{\mathcal{A}, \gamma \vdash^{\mathcal{I}}_{d} E \downarrow \texttt{true} \qquad \mathcal{A}, \gamma \vdash^{\mathcal{I}}_{d} E_1 \downarrow v_1}{\mathcal{A}, \gamma \vdash^{\mathcal{I}}_{d} \texttt{if } E \texttt{ then } E_1 \texttt{ else } E_2 \texttt{ endif} \downarrow v_1}$$

(cond$^{\downarrow}_2$) $$\frac{\mathcal{A}, \gamma \vdash^{\mathcal{I}}_{d} E \downarrow \texttt{false} \qquad \mathcal{A}, \gamma \vdash^{\mathcal{I}}_{d} E_2 \downarrow v_2}{\mathcal{A}, \gamma \vdash^{\mathcal{I}}_{d} \texttt{if } E \texttt{ then } E_1 \texttt{ else } E_2 \texttt{ endif} \downarrow v_2}$$

Definition 5.2.4 specifies the evaluation semantics *direct evaluation* ($\vdash^{\mathcal{I}}_{d}$) which extends Definition 5.1.4 with the rule (a$^{\downarrow}$). This rule states that the evaluation of an attribute for an expression $E.a$ has to return the value $a^{\mathcal{A}}(v)$, i.e. the value which is assigned to attribute a in the context of object v for the current state of $\mathcal{A}$. Consequently, the application of this rule set will always evaluate expressions based on the values stored in $\mathcal{A}$ which in turn is not affected by the evaluation of expressions. It is easy to see that the exclusive application of this definition is pointless as it will never converge in a fixed-point result. However, it represents an integral building block for the methodology detailed in this section as it provides the ability to stop the recursion at an arbitrary point.

Definition 5.2.5 (*Evaluation semantics: direct evaluation*)

The evaluation of an expression E for given attributed object descriptor $(\mathcal{I}, \mathcal{A})$ and an object o conforms to the application of the expression semantics $\vdash^{\mathcal{I}}_{d}$:

$$[\![E]\!]^{d}_{\mathcal{I},\mathcal{A},o} = v \Leftrightarrow \mathcal{A}, \gamma \vdash^{\mathcal{I}}_{d} E \downarrow v$$

Definition 5.2.5 declares that the evaluation of an expression E to a value v conforms to the application of the expression semantics $\vdash_d^{\mathcal{I}}$.

Definition 5.2.6 (*Expression semantics: no recursive calls*)

$$(\texttt{self}^{\downarrow}) \quad \mathcal{A}, \gamma \vdash_n^{\mathcal{I}} \texttt{self} \downarrow \gamma(\texttt{self}) \triangleright \mathcal{A}$$

$$(\mathrm{q}^{\downarrow}) \quad \frac{\mathcal{A}, \gamma \vdash_n^{\mathcal{I}} E \downarrow v \triangleright \mathcal{A}_0 \qquad (\mathcal{A}_{i-1}, \gamma \vdash_n^{\mathcal{I}} E_i \downarrow v_i \triangleright \mathcal{A}_i)_{1 \leq i \leq n}}{\mathcal{A}, \gamma \vdash_n^{\mathcal{I}} E.q(E_1, \ldots, E_n) \downarrow q^{\mathcal{I}}(v, v_1, \ldots, v_n) \triangleright \mathcal{A}_n}$$

$$(\mathrm{a}^{\downarrow}) \quad \frac{\mathcal{A}, \gamma \vdash_n^{\mathcal{I}} E \downarrow v \triangleright \mathcal{A}'}{\mathcal{A}, \gamma \vdash_n^{\mathcal{I}} E.a \downarrow v' \triangleright \mathcal{A}'\{(v,a) \mapsto v'\}} \quad \text{where } \mathcal{A}', \gamma\{\texttt{self} \mapsto v\} \vdash_d^{\mathcal{I}} E_a \downarrow v'$$

$$(\mathrm{cond}_1^{\downarrow}) \quad \frac{\mathcal{A}, \gamma \vdash_n^{\mathcal{I}} E \downarrow \texttt{true} \triangleright \mathcal{A}' \qquad \mathcal{A}', \gamma \vdash_n^{\mathcal{I}} E_1 \downarrow v_1 \triangleright \mathcal{A}''}{\mathcal{A}, \gamma \vdash_n^{\mathcal{I}} \texttt{if } E \texttt{ then } E_1 \texttt{ else } E_2 \texttt{ endif} \downarrow v_1 \triangleright \mathcal{A}''}$$

$$(\mathrm{cond}_2^{\downarrow}) \quad \frac{\mathcal{A}, \gamma \vdash_n^{\mathcal{I}} E \downarrow \texttt{false} \triangleright \mathcal{A}' \qquad \mathcal{A}', \gamma \vdash_n^{\mathcal{I}} E_2 \downarrow v_2 \triangleright \mathcal{A}''}{\mathcal{A}, \gamma \vdash_n^{\mathcal{I}} \texttt{if } E \texttt{ then } E_1 \texttt{ else } E_2 \texttt{ endif} \downarrow v_2 \triangleright \mathcal{A}''}$$

Definition 5.2.6 incorporates $\vdash_d^{\mathcal{I}}$ to specify the *no recursive calls* semantics $\vdash_n^{\mathcal{I}}$ which can derive meaningful judgements for interdependent attributes. As the application of these rules results in the evaluation of attributes and thus a subsequent change in the state of $\mathcal{A}$ (indicated by $\triangleright$), all rules - with the exception of ($\texttt{self}^{\downarrow}$) - update the instantiated attribution (resulting in $\mathcal{A}_n$, $\mathcal{A}'$ and $\mathcal{A}''$ respectively).

Apart from this distinction, the no recursive call semantics employs a modified version of the rule ($\mathrm{a}^{\downarrow}$) to derive a new value for the attribute and to update $\mathcal{A}$ accordingly. Since the respective expression E_a is interpreted using $\vdash_d^{\mathcal{I}}$, this process will not trigger a recursive invocation if a depends on other attributes. In other words, the application of this rule will always substitute attribute references with their current value when evaluating an attribute's expression.

Definition 5.2.7 (*Evaluation algorithm (non-recursive)*)

Expression evaluation for given attributed object descriptor $(\mathcal{I}, \mathcal{A})$ and object o:

$$[\![E]\!]^n_{\mathcal{I},\mathcal{A},o} = [\![E]\!]^d_{\mathcal{I},\mathit{fixpoint}(\mathcal{A}),o}$$

with

$\mathit{fixpoint}(\mathcal{A}) =$

```
do
    for $c_{ctx} \in C, a : c_{ctx} \to c_{ret} = E_a \in A$ do
        for $v \in c_{ctx}^{\mathcal{I}}$ do
            $\mathcal{A}, \gamma\{\texttt{self} \mapsto v\} \vdash^{\mathcal{I}}_n E_a \downarrow v' \triangleright \mathcal{A}'$
            $\mathcal{A} \leftarrow \mathcal{A}'$
        od
    od
    if $\mathcal{A} = \mathcal{A}'$
        return $\mathcal{A}$
od
```

We will now present an algorithm for the fixed-point evaluation of expressions based on the no recursive call semantics. As a prerequisite for the execution of the algorithm shown in Definition 5.2.7, we assume that the initialization values for $\mathcal{A}$ are given semantically as they are not specified in the syntax. In the remainder of this thesis, we employ a syntactical initialization to provide the starting point for the fixed-point computation. To ensure that the algorithm terminates, we request that $\mathcal{A}' \leq \mathcal{A}$ where $\leq$ is the partial order of the semilattice which represents the value domain (cf. Section 2.3.2). The existence of a greatest lower bound and a top element $\top$ guarantees that the algorithm will eventually terminate in a unique fixed-point solution.

Using the semantics of Definition 5.2.6, attribute expressions are evaluated without entering a recursion. This method therefore enables a targeted evaluation of single attribute instances at a given point in the iterative solving process. The downside of this approach is that it is not possible to factor in the dependencies between attributes to guide the evaluation process. Consequently, it is necessary to derive fixed-point results for all attributes in order to determine the final value for a single expression $[\![E]\!]^d_{\mathcal{I},\mathcal{A},o}$. This is realized by the function $\mathit{fixpoint}(\mathcal{A})$ which iterates over all model objects v and triggers the computation of the respective attributes, updating the state of $\mathcal{A}$ accordingly. This process is nested in a `while` loop which ensures that the computation is carried out repeatedly until all values have converged in the most precise fixed-point solution.

5.2.3. Recursive Expression Semantics

We will now present an alternative evaluation semantics for attributes which - in contrast to the method presented in Section 5.2.2 - enables the application of multiple recursive steps in a row. This presents an improvement over the no recursive call method since it enables a focused evaluation of a single attribute by automatically including all of its dependencies in the respective evaluation run.

Definition 5.2.8 (*Expression semantics: recursive calls*)

(self$^{\downarrow}$) $\mathcal{A}, \mathcal{D}, \gamma \vdash_r^{\mathcal{I}} \texttt{self} \downarrow \gamma(\texttt{self}) \triangleright \mathcal{A}, \mathcal{D}$

(q$^{\downarrow}$) $$\frac{\mathcal{A}, \mathcal{D}, \gamma \vdash_r^{\mathcal{I}} E \downarrow v \triangleright \mathcal{A}_0, \mathcal{D}_0 \qquad (\mathcal{A}_{i-1}, \mathcal{D}_{i-1}, \gamma \vdash_r^{\mathcal{I}} E_i \downarrow v_i \triangleright \mathcal{A}_i, \mathcal{D}_i)_{1 \leq i \leq n}}{\mathcal{A}, \mathcal{D}, \gamma \vdash_r^{\mathcal{I}} E.q(E_1, \ldots, E_n) \downarrow q^{\mathcal{I}}(v, v_1, \ldots, v_n) \triangleright \mathcal{A}_n, \mathcal{D}_n}$$

(a$^{\downarrow}$) $$\frac{\mathcal{A}, \mathcal{D}, \gamma \vdash_r^{\mathcal{I}} E \downarrow v \triangleright \mathcal{A}', \mathcal{D}'}{\mathcal{A}, \mathcal{D}, \gamma \vdash_r^{\mathcal{I}} E.a \downarrow a^{\mathcal{A}'}(v) \triangleright \mathcal{A}', \mathcal{D}'} \qquad \text{if } (v, a) \in \mathcal{D}'$$

$$\frac{\mathcal{A}, \mathcal{D}, \gamma \vdash_r^{\mathcal{I}} E \downarrow v \triangleright \mathcal{A}', \mathcal{D}' \qquad \mathcal{A}', \mathcal{D}', \gamma\{\texttt{self} \mapsto v\} \vdash_r^{\mathcal{I}} E_a \downarrow v' \triangleright \mathcal{A}'', \mathcal{D}''}{\mathcal{A}, \mathcal{D}, \gamma \vdash_r^{\mathcal{I}} E.a \downarrow v' \triangleright \mathcal{A}''\{(v, a) \mapsto v'\}, \mathcal{D}'' \cup \{(v, a)\}} \qquad \text{otherwise}$$

(cond$_1^{\downarrow}$) $$\frac{\mathcal{A}, \mathcal{D}, \gamma \vdash_r^{\mathcal{I}} E \downarrow \texttt{true} \triangleright \mathcal{A}', \mathcal{D}' \qquad \mathcal{A}', \mathcal{D}', \gamma \vdash_r^{\mathcal{I}} E_1 \downarrow v_1 \triangleright \mathcal{A}'', \mathcal{D}''}{\mathcal{A}, \mathcal{D}, \gamma \vdash_r^{\mathcal{I}} \texttt{if } E \texttt{ then } E_1 \texttt{ else } E_2 \texttt{ endif} \downarrow v_1 \triangleright \mathcal{A}'', \mathcal{D}''}$$

(cond$_2^{\downarrow}$) $$\frac{\mathcal{A}, \mathcal{D}, \gamma \vdash_r^{\mathcal{I}} E \downarrow \texttt{false} \triangleright \mathcal{A}', \mathcal{D}' \qquad \mathcal{A}', \mathcal{D}', \gamma \vdash_r^{\mathcal{I}} E_2 \downarrow v_2 \triangleright \mathcal{A}'', \mathcal{D}''}{\mathcal{A}, \mathcal{D}, \gamma \vdash_r^{\mathcal{I}} \texttt{if } E \texttt{ then } E_1 \texttt{ else } E_2 \texttt{ endif} \downarrow v_2 \triangleright \mathcal{A}'', \mathcal{D}''}$$

To prevent non-termination in the case of circular dependencies, the rules listed in Definition 5.2.8 introduce the structure $\mathcal{D}$ to identify the point where the evaluation of the expression would enter a cycle. For this purpose, $\mathcal{D}$ keeps track of the attributes (v, a) which have already been processed in the respective derivation. If an attribute has already been evaluated, the first alternative of the rule (a$^{\downarrow}$) applies which - just like the rule (a$^{\downarrow}$) in Definition 5.2.4 - simply retrieves the current value. This approach ensures that the recursive evaluation of cyclic dependencies halts as soon as all attributes belonging to the cycle have been processed once. For attributes which have not yet been "visited" in the current state, the second alternative applies. This rule computes a new value by evaluating E_a, potentially entering a recursion if the expression depends on other attributes. Furthermore, it updates the object descriptor with the new value for the attribute and includes it in the set of recorded dependencies.

Definition 5.2.9 (***Evaluation algorithm (recursive)***)

Expression evaluation for given attributed object descriptor $(\mathcal{I}, \mathcal{A})$ and object o:

```
$[\![E]\!]^r_{\mathcal{I},\mathcal{A},o} =$
            do
              $\mathcal{A}, \gamma \vdash^{\mathcal{I}}_{r} E \downarrow v \triangleright \mathcal{A}'$

              if $\mathcal{A} = \mathcal{A}'$
                return $v$
              fi
              $\mathcal{A} \leftarrow \mathcal{A}'$
            od
```

In this case, the iterative fixed-point evaluation of expressions is controlled by the algorithm shown in Definition 5.2.9. Again, we assume that initialization values are given semantically and that $\mathcal{A}' \leq \mathcal{A}$. Because the evaluation of an attribute automatically triggers the evaluation of all of its dependencies, this approach does not require an outer loop. Consequently, for an expression evaluation $[\![E]\!]^r_{\mathcal{I},\mathcal{A},o}$ for an initial attributed object descriptor $(\mathcal{I}, \mathcal{A})$ and an object o, it is sufficient to evaluate E to v and update the state of the instantiated attribution accordingly. If the new state is equal to the old one, the result is returned. Otherwise, the current state is updated and the computation continues.

5.2.4. Claims

Claim I

We assert that the evaluation of an expression e using the syntax listed in Definition 5.1.3 must yield the same results for (attributed) object descriptors whether the evaluation semantics from Definition 5.1.2 or the algorithms from Definition 5.2.7 or Definition 5.2.9 are used.

$$\mathcal{I}, \gamma \vdash e \downarrow v \Leftrightarrow [\![e]\!]^{d/n/r}_{\mathcal{I},\mathcal{A},o} = v$$

Proof:

It is easy to see that this statement follows from the definitions of the semantics and the evaluation algorithms. Since the primitive form of expressions introduced in Definition 5.1.2 does not include support for data-flow attributes, the algorithms will never enter into an iterative fixed-point computation. The results for the other constituents of expressions are the same.

Claim II

For the recursive call evaluation semantics $\vdash^{\mathcal{I}}_{r}$ we assert that, if an expression E evaluates to v for a given attribution state $\mathcal{A}_1$, then it also evaluates to v for a different state $\mathcal{A}_2$ with the same initial dependency set $\mathcal{D}$ if both states are indistinguishable with respect to the values of the attributes in $\mathcal{D}$ and that subsequently, the resulting states $\mathcal{A}_1'$ and $\mathcal{A}_2'$ will be indistinguishable as well. In other words, if two derivations of E start in similar states, they will consider the same attributes and yield the same result.

This assertion can be formalized as follows:

$$\begin{gathered}\mathcal{A}_1, \mathcal{D}, \gamma \vdash_r^{\mathcal{I}} E \downarrow v \triangleright \mathcal{A}_1', \mathcal{D}' \text{ and } \mathcal{A}_1 \sim_{\mathcal{D}} \mathcal{A}_2 \Rightarrow \\ \mathcal{A}_2, \mathcal{D}, \gamma \vdash_r^{\mathcal{I}} E \downarrow v \triangleright \mathcal{A}_2', \mathcal{D}' \text{ and } \mathcal{A}_1' \sim_{\mathcal{D}'} \mathcal{A}_2'\end{gathered}$$

with $\mathcal{A}_1 \sim_{\mathcal{D}} \mathcal{A}_2$ denoting that $\mathcal{A}_1$ and $\mathcal{A}_2$ are indistinguishable with respect to the attributes in $\mathcal{D}$:

$$\forall v \in c^{\mathcal{I}}, a \in \mathcal{A} : (v, a) \in \mathcal{D} \Rightarrow a^{\mathcal{A}_1}(v) = a^{\mathcal{A}_2}(v)$$

Proof sketch:

Let d be a derivation of E: $d = \mathcal{A}_1, \mathcal{D}, \gamma \vdash_r^{\mathcal{I}} E \downarrow v \triangleright \mathcal{A}_1', \mathcal{D}'$. We perform a structural induction by examining the application of the inference rules $\gamma \vdash \sigma \downarrow \eta$ in a case analysis on the last proof step[1].

$\underline{\sigma = E.a \wedge (v, a) \in \mathcal{D}'}$

Here, the first case of (a$^{\downarrow}$) applies, resulting in:

$$\frac{\mathcal{A}_1, \mathcal{D}, \gamma \vdash_r^{\mathcal{I}} E \downarrow v \triangleright \mathcal{A}_1', \mathcal{D}'}{\mathcal{A}_1, \mathcal{D}, \gamma \vdash_r^{\mathcal{I}} E.a \downarrow a^{\mathcal{A}_1'}(v) \triangleright \mathcal{A}_1', \mathcal{D}'}$$

with the induction hypothesis we can derive the judgement for the right hand side

$$\frac{\mathcal{A}_2, \mathcal{D}, \gamma \vdash_r^{\mathcal{I}} E \downarrow v \triangleright \mathcal{A}_2', \mathcal{D}'}{\mathcal{A}_2, \mathcal{D}, \gamma \vdash_r^{\mathcal{I}} E.a \downarrow a^{\mathcal{A}_2'}(v) \triangleright \mathcal{A}_2', \mathcal{D}'} \text{ and } (v, a) \in \mathcal{D}'$$

In both cases, $(v, a) \in \mathcal{D}'$ and therefore $\mathcal{A}_1' \sim_{\mathcal{D}'} \mathcal{A}_2'$.

$\underline{\sigma = E.a \wedge (v, a) \notin \mathcal{D}'}$

Here, the second case of (a$^{\downarrow}$) applies, resulting in:

$$\frac{\mathcal{A}_1, \mathcal{D}, \gamma \vdash_r^{\mathcal{I}} E \downarrow v \triangleright \mathcal{A}_1', \mathcal{D}' \qquad \mathcal{A}_1', \mathcal{D}', \gamma\{\texttt{self} \mapsto v\} \vdash_r^{\mathcal{I}} E_a \downarrow v' \triangleright \mathcal{A}_1'', \mathcal{D}''}{\mathcal{A}_1, \mathcal{D}, \gamma \vdash_r^{\mathcal{I}} E.a \downarrow v' \triangleright \mathcal{A}_1''\{(v, a) \mapsto v'\}, \mathcal{D}'' \cup \{(v, a)\}}$$

with the induction hypothesis we can derive the judgement for the right hand side

$$\frac{\mathcal{A}_2, \mathcal{D}, \gamma \vdash_r^{\mathcal{I}} E \downarrow v \triangleright \mathcal{A}_2', \mathcal{D}' \qquad \mathcal{A}_2', \mathcal{D}', \gamma\{\texttt{self} \mapsto v\} \vdash_r^{\mathcal{I}} E_a \downarrow v' \triangleright \mathcal{A}_2'', \mathcal{D}''}{\mathcal{A}_2, \mathcal{D}, \gamma \vdash_r^{\mathcal{I}} E.a \downarrow v' \triangleright \mathcal{A}_2''\{(v, a) \mapsto v'\}, \mathcal{D}'' \cup \{(v, a)\}}$$

and $(v, a) \notin \mathcal{D}''$

The claim $\mathcal{A}_1'' \sim_{\mathcal{D}''} \mathcal{A}_2''$ follows from $\mathcal{A}_1''\{(v, a) \mapsto v'\} \sim_{\mathcal{D}'' \cup \{(v,a)\}} \mathcal{A}_2''\{(v, a) \mapsto v'\}$.

[1] Since each nested term is a proper subset of its parent, all derivation trees are of finite height.

6. Model-based Integration of Data-flow Analysis

In the previous chapter, we investigated the formal properties of the DFA method with respect to its application in the technological space of modeling. Based on these definitions, we will now describe in detail the artifacts that are required to specify and execute analyses using standards and technologies from the MDE domain. For this purpose, a dedicated model-based domain-specific language has been developed. This specification language supports the annotation of existing metamodels with analysis constructs in the notion of attribute grammars. The application of the presented methods is illustrated in the context of a running example.

This chapter comprises the following parts: In Section 6.1, we define the abstract syntax of the analysis specification language in the form of a metamodel. A concrete, textual syntax for this model-based DSL is presented in Section 6.2. The process of instantiating a specified analysis for a given model is described in Section 6.3. Data-flow rules can be written in an arbitrary executable language as long as they adhere to the standardized interface required by the DFA solver. This is explored in Section 6.4 and exemplified for the widely-used languages Java and OCL. Finally, in Section 6.5 we demonstrate how a (potentially cyclic) equation system can be derived from an instantiated analysis and how it can be solved by applying fixed-point evaluation semantics.

6.1. Abstract Syntax

In this section, we present a metamodel that implements the abstract syntax of the analysis specification language. It has been designed in the notion of attribute grammars to support the non-intrusive enrichment of existing modeling languages with DFA expressions. The advantage of this approach is that neither the meta language MOF nor the target metamodel have to be modified in any way. The fact that all existing artifacts remain unaware of the annotated analyses ensures that full compatibility with existing standards and practices is preserved. The underlying principles of this approach are discussed in Section 6.1.1 while the actual metamodel is presented in Section 6.1.2.

Furthermore, we introduce a simple modeling language for control-flow graphs that will serve as a running example to demonstrate the application of the developed methods. Section 6.1.3 describes the corresponding metamodel and defines a DFA-based *reachability analysis* to visualize the structural composition of the analysis constructs.

6.1.1. Attribute-based Metamodel Extension

The abstract syntax of the specification language must realize all components required for the definition of a data-flow analysis as well as properly encoding their relationships. In Section 4.2.2, the constructs that are relevant to the analysis specification process were derived from the stated objectives and the properties of the target domain.

However, from a technical perspective, there is another factor that influences the language design which pertains to architectural considerations in the realization of the proposed method. For this purpose, we investigated the relationships between the technological spaces of compiler construction and MDE to assess the impact of the conceptual similarities and differences between these two areas on the transfer of the DFA method to the modeling domain. In this context, we discussed the alignment of abstraction layers (Section 4.1.3) and the principles behind the annotation of constructs at MOF-based metamodels (Section 4.1.4). In Section 4.2.1, we also stressed the importance of maintaining compatibility and ensuring that the analysis remains unintrusive.

The conceptual aspects of analysis specification and execution that will form the basis for the descriptions in this section were outlined in Figure 4.10. Through an evaluation of the attribute grammar technique, we derived a method that implements a corresponding functionality for the purpose of model analysis. For each construct available in attribute grammars, we proposed a respective counterpart in the modeling space. The results were further concretized in Figure 4.11, where we described the resulting elements and their relationships with each other as well as with the classes and objects in the target (meta) models in more detail. In short, the approach can be summarized as follows: Occurrences of attributes are annotated at classes in a metamodel and instantiated for derived model objects while the structural composition of the attribution is itself governed by a model-based specification residing on the *M3* layer.

From a technical view point, this concept presents a problem insofar as a direct implementation would require to extend MOF with the necessary attribution constructs which could then be used to annotate metamodels on *M2*. This process would however require a modification of the MOF language and thus endanger compatibility with existing standards and tooling ecosystems. Since this explicitly contradicts the stated objectives an alternative approach must be devised.

As a straightforward solution to this problem, we can define the abstract syntax not as an extension of the *M3* layer but rather as a model-based domain-specific language or, in other words, a conventional *M2* metamodel. A concrete analysis therefore corresponds to a *M1* model instance of the analysis metamodel. An attribute occurrence contained in this model can then directly reference the class in the target metamodel at which the corresponding attribute should be annotated. Strictly speaking, these connections therefore cross the line between the *M1* abstraction layer, on which the analysis resides, and the *M2* layer on which the target metamodel is defined. This is however not a problem since the MOF modeling framework can specify references between model and metamodel elements. By defining the analysis language itself as a traditional metamodel, we can also make use of MDE tool support to develop the analysis DSL.

This approach has one effect that requires special attention: Since an analysis is now technically a model, it cannot itself be instantiated. To instantiate an analysis, we therefore have to provide a separate modeling language that acts as a container for the instantiated attributes and connects them to the target model.

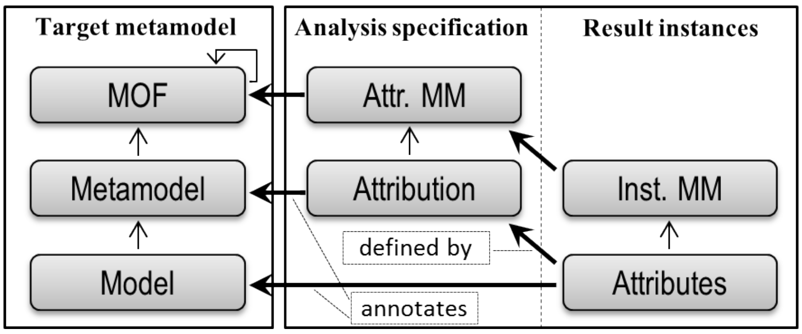


Figure 6.1.: Definition and instantiation of attribute-based metamodel extensions [SB13].

The general idea behind this architecture is illustrated in Figure 6.1. Note that for reasons of clarity and comprehensibility, we have aligned the analysis artifacts according to their conceptual properties rather than placing them on the layer that represents their actual technical type. Consequently, the attribution metamodel AttrMM is located on the top layer because it effectively defines an extension of *M3* although it is actually a *M2* metamodel. Similarly, *M1* attribution models (Attribution) that constitute analysis specifications are located on the same level as the *M2* metamodels that they annotate.

The metamodel that encodes the structure of attribute-based flow analyses will subsequently be referred to as attribution metamodel or AttrMM for short. By defining a reference to the MOF metaclass, it supports the annotation of modeling languages through attribution models. Effectively, this method therefore implements the annotation concept as depicted in Figure 4.11.

The architecture shown in Figure 6.1 already includes a solution to the previously mentioned problem of instantiating attribution models. We can circumvent this issue by simply defining a separate metamodel that formalizes the structural properties of these instances. The metamodel that describes the properties of attribute instances is called instantiation metamodel (or InstMM) and will be discussed in more detail in Section 6.3.1. At this point, it is sufficient to know that it specifies links to both the attribute occurrence class from AttrMM and to MOF's class instance concept (i.e. the metaclass of all model elements). An attribute instance contained in an instantiation model can therefore directly reference the occurrence from which it has been created and the model object at which it is annotated (cf. Section 6.3.1). Each attribute instance conforms to its respective type definition in the form of an occurrence while the object in the target model represents its context. This distinction between specification and instantiation of attributes requires that the logic that governs the creation of attribute instances has to be implemented separately. The instantiation process itself is described in Section 6.3.2.

It should be noted that the use of this model-centric approach is not mandatory. A DFA solver could very well implement the instantiation data structures in an internal, proprietary format without relying on model-based specifications. While this approach would also be viable, the motivation behind the presented method is that it provides a consistent and integrated representation of attributions and their instantiations. All properties, including attribute type definitions and annotation links to the elements in the target (meta) models are encoded in a unified model-based representation.

6.1.2. Attribution Metamodel

We will now give a description of the metamodel that represents the abstract syntax of the analysis specification language. For this purpose, the metamodel implements the language elements that have been proposed in Section 4.2.2 and formally defined in Chapter 5 and specifies their properties and relationships. It has been developed using the EMF framework and therefore conforms to the EMOF subset of the Meta Object Facility.

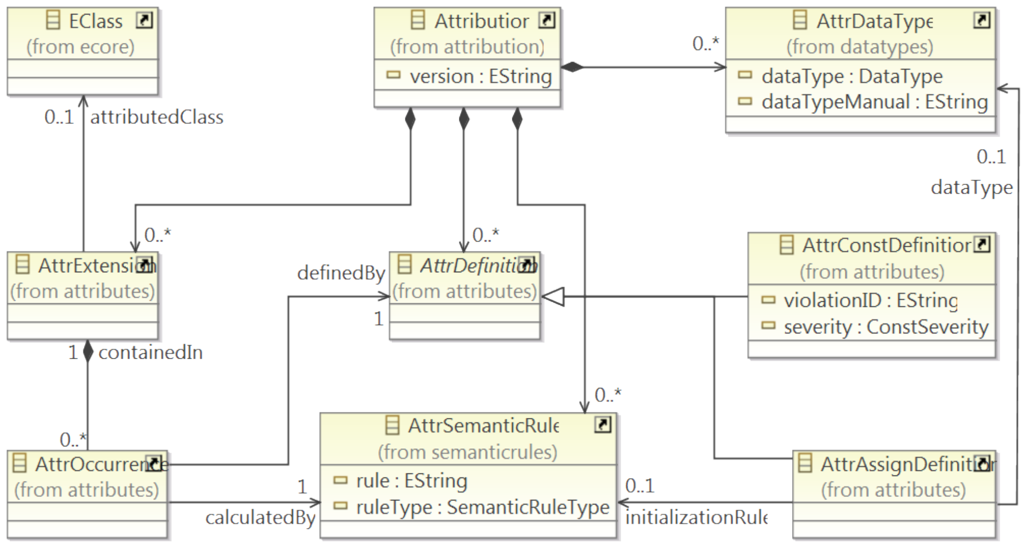


Figure 6.2.: Condensed version of the analysis specification metamodel (AttrMM).

The basic language constructs, i.e. the classes and enumerations, have been grouped into four packages according to their respective responsibilities in the task of assigning attributes to metamodel classes. These packages are: Attribution, Attributes, SemanticRules and DataTypes. A condensed version of the metamodel which aggregates the central concepts from the different packages in a single representation is depicted in Figure 6.2. A full listing of all contents can be found in Appendix B.1.

We will now give a short description of the central elements and their properties:

Attribution

An attribution acts as a container for an arbitrary number of attribution concepts, namely attribute definitions, datatypes, semantic rules and attribute extensions which in turn contain the occurrences. An attribution is itself part of an attribution collection (not present in the condensed version of the metamodel). This structure organizes attributions into libraries from which a DFA solver can then pick and assemble customized collections of analysis functions.

AttrDefinition

An attribute definition represents the type declaration of an attribute. It is an abstract class with the two specializations assignment and constraint.

AttrAssignDefinition

Attributes of the type assignment are able to compute result values of an arbitrary type. For this purpose, they must possess both a datatype (dataType) and a semantic rule which is used to initialize the fixed-point calculation process (initializationRule).

AttrConstDefinition

A constraint can be considered to be an assignment of type boolean whose derived instances are automatically initialized with the value false. In addition to assignments, they define a severity level and a violation message (class attributes violationID and severity). These properties can be used to generate appropriate feedback for the user if the constraint is not fulfilled.

AttrDataType

Datatypes specify the value domain for assignment results. Knowledge about the type of an attribute value is required to generate valid method signatures for languages such as OCL.

AttrSemanticRule

A semantic rule is a function that is executed in the context of an attribute instance and yields a result that represents either the initialization or the iteration value of the instance. In that sense, this concept corresponds to dataflow equations. The semantic rule class provides support for the invocation of methods that are implemented in different languages. This is discussed in more detail in Section 6.4.

AttrExtension

Attribute extensions are containers that connect a set of occurrences to a class in the target metamodel (reference attributedClass). As mentioned in Section 6.1.1, the reference to the target class creates a link between an object in the attribution model and a metamodel class, thereby bridging the *M1* and *M2* layers.

AttrOccurrence

An occurrence of an attribute definition indicates the presence of the respective attribute at a class in the target metamodel. An attribute may have multiple occurrences at different classes. Each occurrence is contained in exactly one attribute extension (containment reference containedIn) and has a reference to an assignment or constraint that constitutes its type declaration (definedBy).

EClass
This element is the EMF counterpart of the MOF metaclass object and therefore represents all classes in the target metamodel. While it is not part of the AttrMM metamodel, it has been included in Figure 6.2 to visualize the link between the attribution metamodel and MOF's *M3* layer.

In some cases, e.g. when an attribution is given in a textual representation (cf. Section 6.2) and has to be parsed, it is important that elements can be addressed via a unique identifier rather than through the references between model objects. For this purpose, the following concepts inherit from the abstract class AttributionElementWithID (omitted from the condensed version of the metamodel): AttributionCollection, Attribution, AttrDefinition, DataType and SemanticRule.

6.1.3. Running Example

To clarify the meaning and the usage of the language constructs presented in the previous section, we will now apply them to a simple use case. The use case is based on the control-flow graph metamodel shown in Section 4.2.2 and will serve as a running example in the following sections. In contrast to the predecessor analysis that was used in Figure 4.12 to exemplify the instantiation process, we now employ a reachability analysis to demonstrate the implementation of a simple validation scenario. In this section, we will focus on the structural layout of the elements that constitute the analysis specification. This includes the relationships between the attribution and the target metamodel which together form an attributed metamodel.

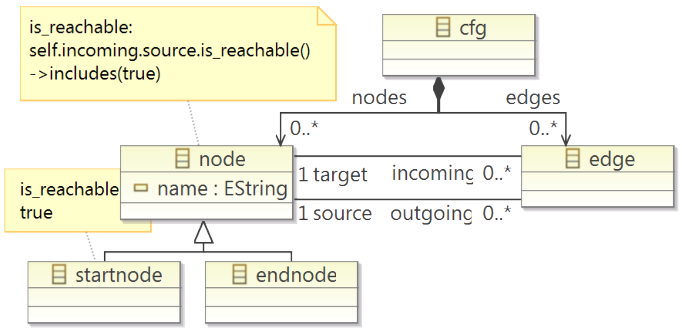


Figure 6.3.: Control-flow graph metamodel with an annotated analysis [SB13].

The target metamodel of the running example is shown in Figure 6.3. It defines a modeling language for control-flow graph structures which can be considered to be very simplistic versions of UML Activity Diagrams. The figure also contains a reachability attribution which is annotated directly at the metamodel's classes. Since the textual format of analysis specifications is the subject of the next section, a simple representation was chosen that is reminiscent of the annotation of OCL constraints.

In all, the metamodel defines six classes with the following semantics: A control-flow graph cfg contains edges and named nodes as its primary constituents. The references between these two concepts connect the graph's nodes through directed edges, thus forming a directed (potentially cyclic) graph. Each node may possess multiple incoming and outgoing edges while an edge has exactly one source and one target node. The beginning and the end of the control-flow are indicated by a startnode and an endnode respectively, both of which are specializations of the node class.

We additionally define the following set of well-formedness rules:

Definition 6.1.1 (***Well-formedness rules for control-flow graphs***)

WF1 The names of all nodes are unique.

WF2 Each graph contains exactly one startnode and one endnode.

WF3 A startnode has at least one outgoing but no incoming edge and vice versa for an endnode.

WF4 Other nodes have at least one incoming and one outgoing edge. All edges have a source and a target node (*connectedness*).

WF5 All nodes are reachable from the startnode (*reachability*).

WF6 The endnode is reachable from all nodes (*liveness*).

It is easy to see that the expressiveness of the OCL is completely sufficient to implement constraints WF1 through WF4. This is however not the case for WF5 and WF6. To determine the reachability/liveness state of each node, the respective state of the immediate predecessor/successor nodes must be known which in turn depends on the state of its predecessors/successors and so on. In other words, the analysis relies on flow-sensitive information.

The reachability state for each node can be easily defined using a recursive specification: While the startnode is always reachable, all other nodes are reachable only if at least one of their immediate predecessors is reachable. Liveness can be computed correspondingly by substituting endnode for startnode and reversing the direction of information flow (backward analysis).

Consider the example model shown in Figure 6.4 in which nodes ③, ④ and ⑤ are not reachable from startnode Ⓢ. The problematic node ③ can be identified easily since it also fails the connectedness property (WF4). However, this is not the case for ④ and ⑤ because of the cyclic control-flow between these elements. To identify these problematic cases, a flow-sensitive analysis is required.

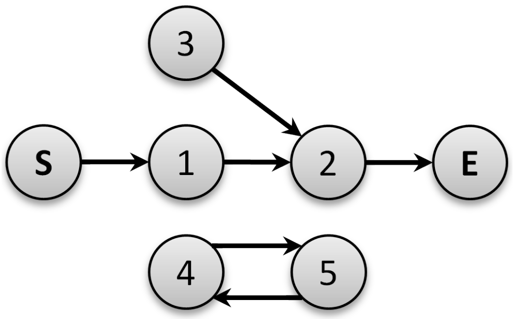


Figure 6.4.: Example control-flow graph model.

The annotation depicted in Figure 6.3 defines a reachability analysis that relies on the existence of an assignment attribute[1] isReachable with the datatype boolean and the initialization value false[2]. A noticeable difference when comparing this analysis to the predecessor analysis from Section 4.2.2 is the presence of two distinct occurrences at the classes node and startnode which nevertheless share the same attribute type isReachable. Taking the semantics of the generalization relationships into account, the occurrence at the node class will also be inherited to the subtypes startnode and endnode. However, since startnode declares its own occurrence of isReachable, the occurrence inherited from node is overwritten. Consequently, instances of isReachable at model objects of the type node and endnode will be calculated with one rule, instances at startnode elements with the other.

The rules for this simple use case have been written in OCL as its navigation expressions allow for short and concise statements. The expression self.incoming.source builds a set of a node's immediate predecessors. From each predecessor, the value of the attached isReachable instance is requested through an invocation of the correspondingly named operation isReachable(). It is important to note that these attribute access operations cannot be implemented through standard function calls as this would lead to an infinite recursion in the case of backward edges. Instead, the responsibility for managing the invocation order for rules and the provisioning of the correct instance values lies with the data-flow solver (cf. Section 6.4 and Section 6.5).

If at least one of the retrieved instance values is true, the returned result for the current node is also true. In summary, we can conclude that this specification is a direct implementation of the declarative definition of the reachability property which has been presented above.

Figure 6.5 shows an abstract representation of the attributed metamodel with a focus on the structural composition of the attribution constructs. The diagram depicts the elements of the attribution model and the classes from the target metamodel at which the occurrences are annotated.

In this diagram, it can be seen that the two occurrences of the assignment attribute isReachable are attached to the startnode and node classes from the target

[1]Section 6.2.2 demonstrates how this assignment can be complemented by a constraint which generates an appropriate feedback message highlighting the problematic element(s).

[2]The declarations of this attribute and its datatype are not shown in the diagram.

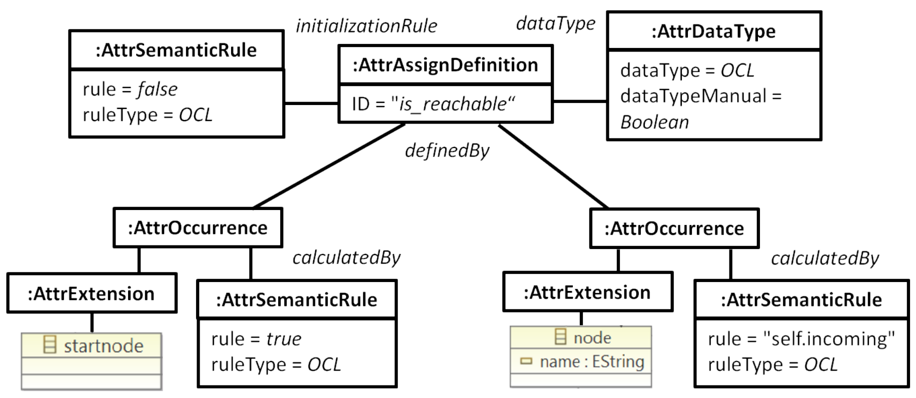


Figure 6.5.: Structure of the abstract syntax of the attributed metamodel depicted in Figure 6.3 [SB13].

metamodel via extension containers. This representation illustrates how this approach manages to ensure a clear separation between an existing modeling language and the components of the attached analysis.

6.2. Concrete Syntax

In the last section, we introduced the attribution metamodel which constitutes the abstract syntax of the analysis specification language. For a practical application of this model-based DSL, we still have to devise a concrete syntax by assigning syntactical representations to the language constructs.

This process requires a decision on the type of the language. Typically, models are represented as diagrams or in a textual format depending on the properties of the language and its intended use. One option would be to visualize attributes as elements connected to the classes in the target metamodel's diagram. Since MOF already defines a commonly used graphical syntax for metamodels, an extension of metamodel diagrams with attribution elements could therefore resemble the visualization of OCL expressions. Another method would be the definition of a textual DSL that uses cross-model links to attach analysis specifications to metamodel classes.

Based on an examination of the design goals and challenges listed in Section 4.2, it has been decided that only the second approach, a text-based language, can fulfill the requirements of our application scenario. In detail, this choice has been motivated by the following considerations:

- The attribution metamodel contains only a small number of core concepts whose relationships define a very specific structure for language expressions. It is therefore possible to develop a simple and concise textual representation of these elements which is easy to learn and can be understood even by someone who is not familiar with the syntax. Support for inexperienced users may be provided through features such as syntax highlighting and autocompletion.

- In many respects, working with text documents simplifies the development process. For example, the version control systems that are used to manage program code are also able to support collaborative work on other types of textual artifacts. Because metamodels and analysis annotation models are separate entities, these artifacts can also be worked on independently.

- Using default text editing functions, textual specifications can be easily split and combined, enabling a modular approach to analysis management. In this context, we can envision an extension of the presented technique that supports the implementation of complex libraries of analysis functions. Similar to the class concept employed in object-oriented programming languages, an analysis could subclass existing specifications and thereby automatically inherit their elements or overwrite them with more specialized definitions.

- A graphical annotation of metamodels would require an extension of existing MDE tooling environments. Since we explicitly stated full compatibility with existing tools as a central design goal, this is not a viable option. It would also contradict the previous point of managing metamodels and attributions separately since the metamodel's diagram would also have to include the attribution elements.

- Because of the structural properties of attributions, it is inherently difficult to specify them using a graphical syntax. Elements such as attribute definitions or semantic rules constitute global definitions which can be referenced and reused in different contexts. It is therefore not clear where these elements should be displayed in a metamodel diagram.

In order to properly define a domain-specific language, it is not sufficient to just assign concrete representations to the language constructs. A clear mapping between the elements of the abstract and the concrete syntax has to be provided as well. The grammar presented in Section 6.2.1 not only defines the concrete syntax but also links the textual representations to the respective classes in the attribution metamodel. In Section 6.2.2, we use the running example from the last section to illustrate how analyses can be formalized in the proposed language.

6.2.1. Specification Syntax

To realize the attribution DSL, the facilities of the Xtext language workbench have been used. The basic principles and techniques behind this toolset are described in more detail in Section 7.1.4. Aside from being able to automatically generate parsers that transform textual expressions into model-based abstract syntax trees, it also supports cross-references between (model) objects, a feature not found in traditional compiler tools based on context-free languages.

Production rules in Xtext are written in a notation that is similar to the EBNF with additional constructs for encoding cross-reference relationships and specifying mappings to metamodel elements.

Standard rules[3] are written using the following syntax:

```
<production rule name> returns <metamodel class>:
  '<text>'
  <class attribute> = <lexer rule>
  <containment reference> += <production rule>
  <cross reference> += [<metamodel class>] ;
```

The semantics of these elements can be described as follows:

- `<production rule name>` is the name by which a rule can be referenced inside the grammar. By convention, we use the name of the return type (see below).
- **returns** `<metamodel class>` determines the type of the model object that will be returned by this rule.
- `<text>` that is part of the source code is enclosed in apostrophes.
- `<class attribute>` values are parsed using lexer rules such as `ID` or `STRING`.
- `<containment reference>` delegates to another production rule to recursively parse contained text fragments and organize the resulting model elements in a containment hierarchy.
- `<cross reference>` sets the value of a cross-reference in the model by specifying the type of the target element that can be referenced.
- Additional operators can be applied to parser instructions to indicate optional constructs (`?`), determine multiplicity (`+`, `*`) and allow unordered groups (`&`).

The concrete syntax incorporates all relevant artifacts of the specification language: Attribute definitions, attribute extensions, semantic rules and datatypes. Except the extensions (and the therein contained occurrences), all elements of an attribution can be cross-referenced. The arrangement of the production rules in the following descriptions reflects the package structure of the AttrMM metamodel.

Attribution Package

Each AttrMM model has exactly one attribution collection object which represents the model's root element:

```
AttributionCollection returns attribution::AttributionCollection:
 ( imports+=Import )*
 ( attributions+=Attribution )* ;
```

Collections serve as containers for imports and attributions and do not possess an explicit syntactical representation themselves. Instead, the existence of a collection is assumed for each document.

```
Import returns attribution::Import:
 'import' importURI=STRING ';' ;
```

[3]A standard rule returns an instance of the specified metamodel class. Enumeration types can be encoded using dedicated `enum` rules.

Imports are used to reference the target metamodels, i.e. the abstract syntax of the modeling language that should be enriched with analysis annotations. While not strictly required by DFA solvers[4], Xtext relies on these import statements to properly resolve cross-references between attribute extensions and classes in the target metamodel.

The second kind of element that can be part of an attribution collection are the attributions themselves:

```
Attribution returns attribution::Attribution:
 'attribution' id=ID
 '{'
    ( 'name' name=STRING ';' )?
    ( 'version' version=STRING ';' )?
    ( 'description' description=STRING ';' )?
    ( ( attrDefinitions+=AttributeDefinition )* &
      ( attrSemanticRules+=SemanticRule )* &
      ( attrDataTypes+=AttrDataType )* &
      ( attrExtensions+=AttrExtension )* )
  '}' ;
```

An element of this type starts with the keyword `attribution` followed by a (unique) identifier. Through this identifier, a user can constrain the fixed-point computation to the attributes contained in specific attributions. Additional meta information can be encoded in optional tags. These fields - name, version and description - are part of the AttributionElementWithID type (cf. Section 6.1.2) and therefore also available at other elements. Statements for attribute definitions, semantic rules, datatypes and extensions are also contained in the body of the attribution element and can be specified in an arbitrary order.

Attributes Package

This package handles the specification of attributes and occurrences and the assignment to classes in the target metamodel.

Using the following syntax, assignment and constraint attributes can be declared:

```
AttributeDefinition returns attributes::AttrDefinition:
 'attribute' ( AttrAssignDefinition | AttrConstDefinition ) ;

AttrAssignDefinition returns attributes::AttrAssignDefinition:
 'assignment' id=ID ( name=STRING )? ( '[' description=STRING ']' )? ':'
   dataType=[datatypes::AttrDataType]
   'initWith' initializationRule=[semanticrules::AttrSemanticRule] ';' ;

AttrConstDefinition returns attributes::AttrConstDefinition:
 'constraint' id=ID ( name=STRING )? ( '[' description=STRING ']' )? ':'
   severity=ConstSeverity
   ( violationID=STRING )? ';' ;

enum ConstSeverity returns attributes::ConstSeverity:
 info | advice | warning | error;
```

For assignments, the declaration has to start with the keyword `attribute` followed by `assignment`. For constraints, the term `constraint` is used instead. In each case, the second keyword instructs the parser to interpret the following characters using

[4]The required artifacts may also be directly configured inside the solver module (cf. Section 8.3).

the correct production rule for the respective type. Here, the grammar directly reflects the structure of the attribution metamodel with an abstract element and two concrete specializations.

A unique identifier is required for both attribute types while the meta information tags are again optional. The remaining part of the declaration syntax is different for both elements. Each assignment has to define cross-references to a datatype and to an initialization rule. Constraints, on the other hand, possess a severity level and an optional message that can be shown to the user if the check fails.

Defining occurrences and binding them to classes is now simply a matter of establishing the correct relationships between the existing elements:

```
AttrExtension returns attributes::AttrExtension:
  'extend' attributedClass=[ecore::EClass]
    'with' '{' (attributes += AttrOccurrence)* '}' ;

AttrOccurrence returns attributes::AttrOccurrence:
  'occurrenceOf' definedBy=[attributes::AttrDefinition]
    'calculateWith' calculatedBy=[semanticrules::AttrSemanticRule] ';' ;
```

An attribute extension starts with the keyword extend followed by a reference to a class from the target metamodel. In its body, an extension lists the occurrences which should be attached to this class. Each occurrence is characterized by an attribute type and a semantic rule that computes its data-flow iteration values. Meta tags are not available for these elements since they merely point to existing concepts which have been defined elsewhere.

SemanticRules Package

All semantic rules conform to a single class type in the attribution metamodel. However, different methods for executing the corresponding code also require different parameters. To implement this property, the rules governing the interpretation of these statements employ the same principle used for parsing attribute definitions: A single production rule serves as starting point and redirects the parser to the matching subtype. Currently, the types java, ocl and auto are supported.

```
SemanticRule returns semanticrules::AttrSemanticRule:
  'rule' (SemanticRuleJava | SemanticRuleOCL | SemanticRuleAuto) ';' ;
```

The keyword java indicates that the rule is written in the Java language.

```
SemanticRuleJava returns semanticrules::AttrSemanticRule:
  'java' id=ID ( name=STRING )? ('[' description=STRING ']')? ':'
    ruleType=SemanticRuleTypeJava
    rule=STRING ;

enum SemanticRuleTypeJava returns semanticrules::SemanticRuleType:
 java_call='call' | java_constructor='constructor' | java_inline='inline' ;
```

The enumeration SemanticRuleTypeJava defines multiple techniques that can be used to execute Java code. A call invokes a method using the name specified in the rule field while constructor instantiates the class of the given name. Finally, by specifying the ruleType as inline, Java source code can be directly embedded in the rule's definition (cf. Section 6.4.3).

OCL expressions may be written in either the standard or in the imperative form:

```
SemanticRuleOCL returns semanticrules::AttrSemanticRule:
 'ocl' id=ID ( name=STRING )? ('[' description=STRING ']')? ':'
   ruleType=SemanticRuleTypeOCL
   rule=STRING ;

enum SemanticRuleTypeOCL returns semanticrules::SemanticRuleType:
 ocl='standard' | impocl='imperative' ;
```

Finally, the type auto provides the null element as well as a default value constant which can be used to initialize the fixed-point computation (by interpreting this value as the top element $\top$, cf. Section 9.2 for a demonstration of how this INIT constant can be employed in practice):

```
SemanticRuleAuto returns semanticrules::AttrSemanticRule:
 'auto' id=ID ( name=STRING )? ('[' description=STRING ']')? ':'
   ruleType=SemanticRuleTypeAuto ;

enum SemanticRuleTypeAuto returns semanticrules::SemanticRuleType:
 const='constant' | null='null' ;
```

DataTypes Package

The definition of the syntax for datatypes follows a similar pattern. A datatype declaration starts with type followed by an additional characterization that distinguishes between different kinds of datatypes.

```
AttrDataType returns datatypes::AttrDataType:
 'type' (AttrDataTypeJava | AttrDataTypeOCL) ';' ;
```

If an attribute uses Java objects to represent its values, it must be assigned a corresponding datatype:

```
AttrDataTypeJava returns datatypes::AttrDataType:
 'java' id=ID ( name=STRING )? ('[' description=STRING ']')? ':'
   dataType=DataTypeJava;

enum DataTypeJava returns datatypes::DataType:
 Java_Object='object';
```

Datatypes for OCL are based on the OCL typing system. The most commonly used types are encoded in the enumeration DataTypeOCL and can be directly referenced. Alternatively, it is possible to set the definition to manual and provide a custom declaration.

```
AttrDataTypeOCL returns datatypes::AttrDataType:
 'ocl' id=ID ( name=STRING )? ('[' description=STRING ']')? ':'
   (dataType=DataTypeOCL | dataType=DataTypeOCLManual dataTypeManual=STRING);

enum DataTypeOCL returns datatypes::DataType:
 OCL_Integer='integer' | OCL_Boolean='boolean' | OCL_Real='real' |
 OCL_String='string' | OCL_Set_String_='set(string)' | OCL_Set_Integer_='set(integer)' |
 OCL_Set_Any_='set(any)' | OCL_OrderedSet_String_='orderedset(string)' |
 OCL_OrderedSet_Integer_='orderedset(integer)' | OCL_OrderedSet_Any_='orderedset(any)' |
 OCL_Sequence_String_='sequence(string)' | OCL_Sequence_Integer_='sequence(integer)' |
 OCL_Sequence_Any_='sequence(any)' | OCL_Bag_String_='bag(string)' |
 OCL_Bag_Integer_='bag(integer)' | OCL_Bag_Any_='bag(any)';

enum DataTypeOCLManual returns datatypes::DataType:
 OCL_ManualDeclaration='manual';
```

6.2.2. Running Example

We will now demonstrate the usage of the presented domain-specific language in the context of a use case. For this purpose, we revisit the running example that has been introduced in Section 6.1.3.

Algorithm 3 The attribution reachability_analysis

```
attribution reachability_analysis
    – attribute that indicates whether a node is reachable from the start node
    attribute assignment isReachable : OCLBoolean
        initWith boolean_false;
    – ocl rule to check whether at least one of the direct predecessors is reachable
    rule ocl node_isReachable : standard
        "self.incoming.source.isReachable()->includes(true)";
    – attach 'isReachable' to 'nodes' and compute with rule 'node_isReachable'
    extend node with {
        occurrenceOf isReachable
            calculateWith node_isReachable;
    }
    – attach 'isReachable' to 'startnodes' (overwriting the 'node' occurrence) and always set to true
    extend startnode with {
        occurrenceOf isReachable
            calculateWith boolean_true;
    }
}
```

Using the textual domain-specific language, we are able to give a full specification of the reachability analysis as shown in Algorithm 3.

Algorithm 4 The attribution reachability_validation

```
attribution reachability_validation
    – constraint that indicates whether a node is reachable
    attribute constraint checkReachability : error
        "unreachable node detected";
    – use the result of attribute 'isReachable'
    rule ocl node_checkReachability : standard
        "self.isReachable()";
    – attach 'checkReachability' to 'nodes' and compute with rule 'node_checkReachability'
    extend node with {
        occurrenceOf checkReachability
            calculateWith node_checkReachability;
    }
}
```

The constraint checkReachability specified in the attribution listed in Algorithm 4 processes the results computed for isReachable instances to provide corresponding feedback to the user.

In Section 4.2.2, the control-flow graph modeling language has been used to illustrate some basic properties of the data-flow evaluation process. The example given in that context computes the set of all (transitive) predecessors for each of the graph's nodes. To further clarify the representation of the attribution concepts in the proposed syntax, we enhance this definition with a second analysis that determines whether a node is part of a strongly connected component (SCC). A strongly connected component is a maximal cyclic subgraph of a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ in which, for each pair of vertices $v, w \in \mathcal{V}$, there exist paths $p_1 : v \overset{*}{\Rightarrow} w$ and $p_2 : w \overset{*}{\Rightarrow} v$ [Tar72].

This goal can be achieved in a number of ways. By reusing the results of the predecessor analysis, it is possible to check whether a node is a predecessor of itself. Another method would be to compare the result set of each node to the union of the results at its direct predecessors to see whether both sets contain the same elements. In each case, the test reveals whether a node is part of a cyclic structure while the data-flow mechanism ensures that the final result will consist only of the maximal cycles.

Since a control-flow can contain multiple SCCs, a further characterization is required which uniquely identifies the SCC the respective node belongs to. By definition, each node can only be part of exactly one SCC and all of its members share the same predecessors. If it is determined that a node belongs to a SCC, a matching identifier can therefore be generated by simply mapping the node's transitive set of predecessors to a single value, e.g. using a hash function.

Algorithm 5 The attribution flow_analysis

```
attribution flow_analysis

    – the set of all predecessors
    attribute assignment allPredecessors : OCLSet
        initWith set_empty;

    – identifies the SCC the node belongs to (or '0' otherwise)
    attribute assignment sccID : OCLInteger
        initWith int_zero;

    – create union of direct predecessors and transitive predecessors at the direct predecessors
    rule ocl node_allPredecessors : imperative
        "var directPred : Set(node) := self.incoming.source->asSet();
         var transitivePred : Set(node) := self.incoming.source.allPredecessors()->asSet();
         return directPred->union(incomingPred)";

    – compare local 'allPredecessors' to predecessors' 'allPredecessors' and generate identifier
    rule ocl node_sccID : imperative
        "var selfPred : Set(node) := self.allPredecessors()->including(self);
         var allPred : Set(node) := self.incoming.source.allPredecessors()->asSet();
         if (allPred=selfPred) then
            return selfPred->hashCode()
            else return 0 endif;";

    – alternative method to detect SCC membership: check if node is predecessor of itself
    rule ocl node_sccID_alternative : standard
        "if (self.allPredecessors()->contains(self)) then
            return self.allPredecessors()->hashCode()
            else return 0 endif;";

    – attach 'allPredecessors' and 'sccID' to 'node'
    extend node with {
        occurrenceOf allPredecessors
            calculateWith node_allPredecessors;

        occurrenceOf sccID
            calculateWith node_sccID;
    }
}
```

The attribution shown in Algorithm 5 defines the attributes allPredecessors and sccID to implement both the predecessor and the SCC analysis:

Note that there are multiple ways to structure analyses. The examples in this section have been split into three attributions: reachability_analysis, reachability_validation and flow_analysis. Ultimately, the decision on which elements should be part of which attribution is up to the developer. This way, both the specification and the evaluation process can be organized and modularized.

For example, reachability_analysis and reachability_validation could both be part of the same attribution collection (i.e. be located in a single text document). Alternatively it would also be possible to combine the definition of isReachable and checkReachability in a single attribution or even split them across separate files. Regardless of the chosen method, once the attribution(s) have been loaded by a DFA solver[5], the user is able to choose whether only the reachability computation should be carried out or if the validation that generates the constraint messages should be performed as well[6].

For the presented flow_analysis attribution, it could also make sense to move the predecessor computation into a separate attribution which could then be reused as a module by other analyses that depend on its results.

6.3. Analysis Instantiation

Once an analysis has been defined for a metamodel, it can be instantiated for derived models. This process is a prerequisite for the execution of the fixed-point computation. In this section, we detail the principles behind the instantiation of flow analysis attributions and the necessary steps to carry out this task.

In the first step, we describe the instantiation metamodel. This specification encodes the structure of attribute instances and is the topic of Section 6.3.1. In Section 6.3.2, we examine basic properties of the instantiation process while the corresponding algorithms are listed in Section 6.3.3. Like in the previous sections, we will demonstrate the application of the presented techniques in the context of the running example. This is the subject of Section 6.3.4.

6.3.1. Instantiation Metamodel

In Section 6.1.1 we motivated and detailed an architectural layout for the technical implementation of the analysis technique. An integral part of this concept is the partitioning of the analysis language. To avoid any modification to the MOF's *M3* layer, two different metamodels are used to describe the structure of attribute specifications and instantiations respectively. The attribution metamodel AttrMM that implements the abstract specification syntax has been presented in Section 6.1.2. Now, we will focus on the second part, the instantiation metamodel InstMM.

This artifact provides a container for instantiated attributions by encoding the structural properties of attribute instances as well as their relationships to all of the

[5] Section 6.3.3 describes how multiple attributions can be merged.

[6] Techniques for the configuration and customization of analyses are discussed in Section 7.3.1. The Model Analysis Framework employs the Evaluation Strategy concept to constrain the evaluation process to a relevant subset of attributes.

involved modeling artifacts. This encompasses the instance's defining occurrence from the attribution model and the context object from the target model. By tying together the related concepts, this definition realizes a sound, model-based representation of these elements.

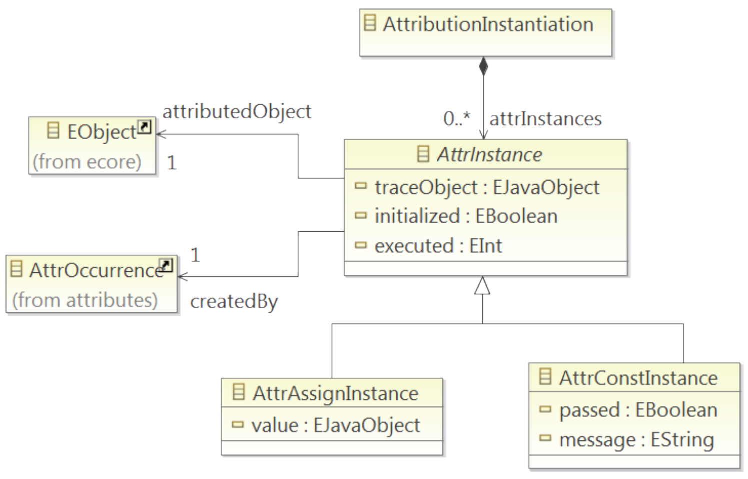


Figure 6.6.: Analysis instantiation metamodel (InstMM).

The instantiation metamodel InstMM is shown in Figure 6.6. Its elements and their properties can be characterized as follows:

AttributionInstantiation
This is the root object that contains all attribute instances (attrInstances) that have been created for the target model during the instantiation process.

AttrInstance
This abstract class represents a single instantiated attribute attached to a specific object in the target model. The associated model element is referenced through the attributedObject relationship while createdBy connects the instance to its defining occurrence from the attribution model. Since the occurrence is itself connected to an attribute definition, this link implicitly provides the attribute type of the instance. The class in the target metamodel at which the occurrence has been annotated is also available through this reference.

The class attributes initialized and executed can be used by fixed-point algorithms to store object-dependent data during the solving process. The field traceObject can provide an alternative representation of a model object.

AttrAssignInstance
Assignment instances represent instantiated occurrences of attribute assignments. Consequently, the value field holds a value from the attribute's datatype domain. Its contents may change during the evaluation process as it is first set to the initialization value and then updated with new fixed-point iteration results.

AttrConstInstance
A constraint instance represents an instantiation of a constraint occurrence. The field passed indicates whether the validation of the constraint was successful. The message string can be modified by data-flow rules to provide customized problem indicators that can be displayed to the user afterwards (not to be confused with the constant violationID of the constraint class).

EObject
The EObject class from the Ecore meta metamodel is the generic type of all elements in target models. In the instantiation metamodel, it represents the context element of the attribute instance, i.e. the model element to which the instance has been attached.

AttrOccurrence
The occurrence is part of the attribution and denotes the type of an instance.

6.3.2. Instantiation Process

Instantiation refers to the process in which one or more[7] attributions that extend a single target metamodel are instantiated for a given model. In this phase, an *attribute instance* is created for each *attribute occurrence* annotated at the class type of a model element. In the following paragraphs, we give a description of the required input, the instantiation process itself and the resulting artifacts.

Instantiation Artifacts

Instantiation requires three different types of interconnected artifacts as input:

1. The target metamodel, i.e. the abstract syntax of a modeling language.
2. One or more attributions that have been defined for this metamodel.
3. A model conforming to the target metamodel.

Generally speaking, the output of the instantiation process consists of a set of attribute instances associated with elements in the target model. Their obvious role in the execution of an analysis is to hold the data-flow values. However, since the instantiation metamodel establishes connections between the different model artifacts, they also encode detailed information about the relationship between the analysis specification and the target modeling language.

To enable a better understanding of the properties of attribute instances, we can visualize the relationships between the relevant modeling artifacts in a matrix: Figure 6.7 provides a conceptual overview depicting how the elements from the three input artifacts are all tied together by the generated instances. Essentially, the instantiation semantics describe how to transfer attribution annotations

[7]If multiple attributions are provided, they can be merged into a single attribution before the instantiation process is carried out. This is addressed in Section 6.3.3.

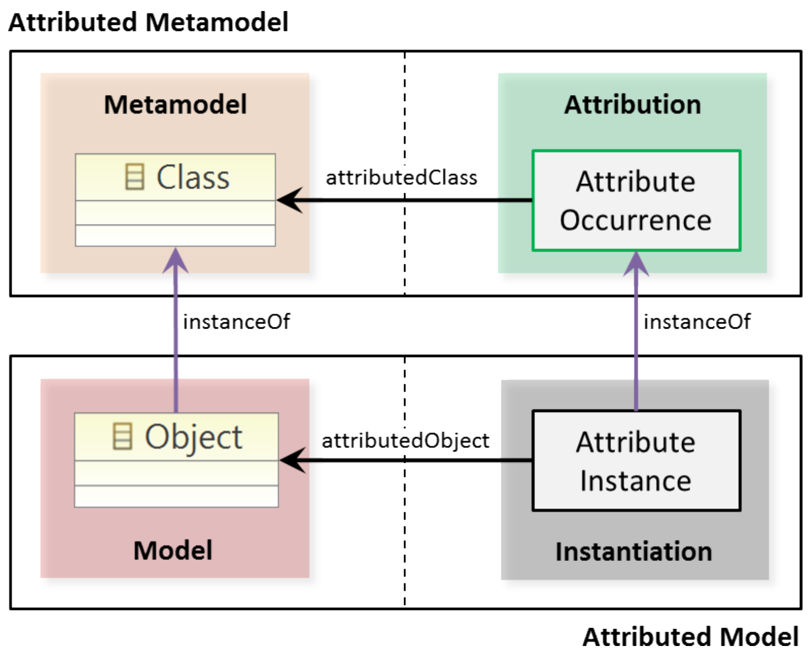


Figure 6.7.: Relationships between model and attribution concepts.

(attributedClass) to the model level (attributedObject), taking into account the generalization hierarchy between classes in the target metamodel.

It is worth mentioning that the four model artifacts can be classified according to different criteria:

1. With respect to the instantiation process, a distinction can be made between input (metamodel, attribution, model) and output (instantiation) artifacts.

2. It is also possible to distinguish between the (conceptual) abstraction layers: The *attributed metamodel* consisting of the attribution and the target metamodel is located on the language definition layer. On the other hand, the model together with its associated instantiation (the *attributed model*) are both expressions that conform to these syntactical definitions.

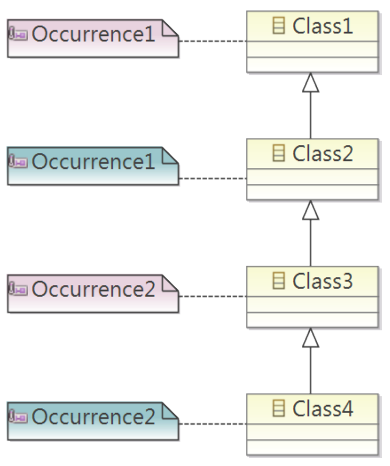
3. As a variation of the previous point, the artifacts can also be classified on the basis of their technical properties. While an attribution effectively constitutes an abstract syntax, its actual format is that of a *M1* model. The target model and the instantiation are also *M1* artifacts while the target metamodel resides on the *M2* layer.

4. Finally, the artifacts can be grouped according to their respective function. Metamodels and models represent the syntactical components of modeling languages while attributions and instantiations constitute analyses.

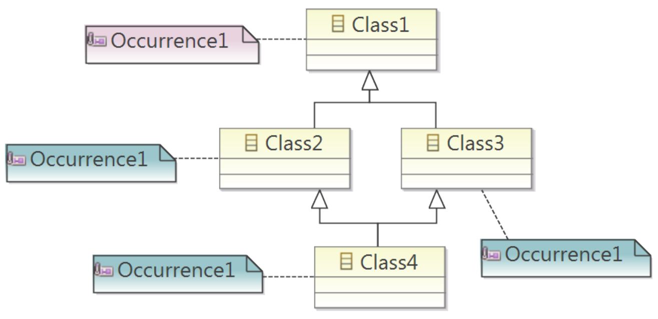
Inheritance Semantics

In order to provide a consistent integration with MDE concepts, the instantiation semantics for analyses has to follow the same guiding principles that are also applied to models. It is therefore clear that an occurrence of an attribute at a metamodel class has to result in an instance at model elements of this specific type. If the target class is part of a generalization hierarchy, this is also true for subclass elements as they implicitly inherit the class type of their parents.

Another important aspect in this context is the possible redefinition of attributes. In certain cases, it is necessary to compute instances at subtypes with a different rule than instances of the same attribute at a parent class. For this purpose, an occurrence of a specific attribute attached to a class may be overwritten at a child class with another occurrence of the same attribute type. An example for this can be found in the reachability analysis where the attribute isReachable is redefined at the startnode class. An instance of an attribute must therefore always be derived from the occurrence annotated at the element's most specific (super)class.



(a) Inheritance with redefinition



(b) Diamond-shaped inheritance

Figure 6.8.: Inheritance of attribute occurrences with redefinition.

The examples shown in Figure 6.8 illustrate this principle. In the diagrams, occurrences that have been specified in the attribution are shown in red while the occurrences whose presence is implicitly assumed based on application of the generalization semantics are marked green.

As indicated in Figure 6.8(a), the redefinition of attributes is straightforward if the metamodel does not contain multiple inheritance. In this example, two occurrences of the same attribute have been attached to Class1 and Class3. The occurrence annotated at Class1 is inherited by its child Class2. At Class3, this element is overwritten by another occurrence which is then again inherited by Class4.

Figure 6.8(b) demonstrates the behavior in the case of a diamond-shaped generalization hierarchy. It comes as no surprise that Occurrence1 is then inherited by both Class2 and Class3. However, an interesting situation arises at Class4 as this element now obtains two occurrences of the same attribute from its parents. This does not present a problem though as both elements originate from the same occurrence attached to Class1. We can conclude that conflicts that arise from multiple inheritance can be resolved for occurrences which share the same point of origin. Consequently, it would also not be a problem if the root element Class1 was removed entirely from the diagram and Occurrence1 was instead attached to *either* Class2 *or* Class3.

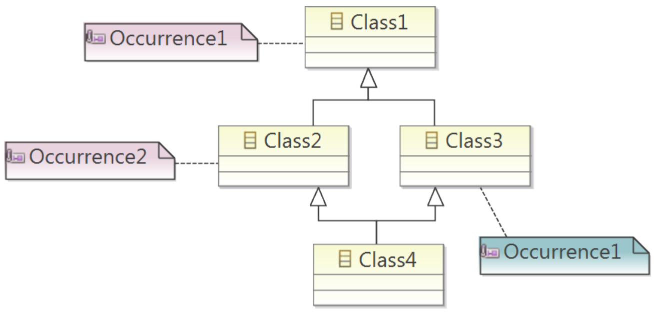


Figure 6.9.: Diamond-shaped inheritance with unclear semantics.

Unfortunately, it is not always possible to resolve these conflicts because certain circumstances prevent the identification of a unique source occurrence. This is an inherent problem with multiple inheritance and also exists in object-oriented programming languages such as C++ (*ambiguous base class*).

The problem is illustrated in Figure 6.9. Again, the metamodel contains a diamond-shaped generalization hierarchy with an occurrence attached to the root element. Since the attribute is now redefined at Class2, Class4 inherits two different occurrences of the same attribute. This results in undefined behavior as instances associated with Class4 objects would now be associated with two different semantic rules[8]. The same would be true if Class1 was removed and Occurrence1 instead had been attached to Class3. Since this situation cannot be meaningfully resolved, this is a violation of the well-formedness rules. This case can be detected through a static validation of the attribution specification and subsequently be indicated to the user.

Note that if an additional occurrence Occurrence3 was attached to Class4, the specification would be nevertheless be valid. In this case, the conflicting occurrences Occurrence1 and Occurrence2 would be overwritten and a new, unique declaration would be available.

6.3.3. Attribute Instantiation

We now present algorithms in pseudo code that implement the instantiation process.

Merging Attributions

To simplify the following steps, multiple attributions contained in the analysis artifacts can be merged. This can be accomplished through a combination of their contents as shown in Algorithm 6.

[8]In practice, it would not be possible to instantiate attributes at Class4 objects at all, since the syntax of InstMM allows only one defining occurrence.

Algorithm 6 Merging attributions

```
1: function MERGEATTRIBUTIONS(Set<Attribution> inputAttributions)
2:     mergedAttribution = new Attribution
3:     for all (inputAttribution : inputAttributions) do
4:         mergedAttribution.definitions += inputAttribution.definitions
5:         mergedAttribution.datatypes += inputAttribution.datatypes
6:         mergedAttribution.semanticrules += inputAttribution.semanticrules
7:         mergedAttribution.extensions += inputAttribution.extensions
       return mergedAttribution
```

Computing available Attribute Occurrences

Before attribute instances can be created for a model object, it must first be determined which occurrences are available for the object's class type. To provide proper support for attribute inheritance and the redefinition of attributes at subtypes, it is necessary to evaluate the generalization relationships in the metamodel.

A simple approach to compute the set of occurrences for a model object would consist of iterating over its class hierarchy to identify all matching occurrences. Starting with the element's concrete class, all occurrences in the attribution that have been assigned to this type are added to a set. Through a traversal of the generalization hierarchy, this collection can then be extended with occurrences defined at parent classes. Redefinition semantics demand that only the most specific occurrence of an attribute is kept. Occurrences at superclasses may therefore only be added if an overriding occurrence of the same attribute is not already present in the set.

This behavior is implemented in Algorithm 7. The function collectOccurrences returns the occurrences which are available at a specific model object. For this purpose, the method addOccurrences traverses the generalization hierarchy upwards to collect occurrences assigned to parent classes. In line [3], the occurrences for the respective class are read from the attribution. Lines [7-11] check whether an occurrence of the same attribute type is already present which would constitute a redefinition of this attribute. If this is not the case, it is added to the collection of occurrences for the model element [13-14]. Finally, this process is recursively invoked for the respective parents [16-17].

Static Enhancement of the Attribution

It is obvious that this method is inefficient since the traversal of the generalization hierarchy has to be repeated for each object. To alleviate this problem, occurrence availability can be computed once for all classes in the target metamodel and kept in an internal data structure. We propose an alternative method that implements this approach by storing the generated information in the abstract syntax of the attribution language. The basic idea is to generate a new attribution that enhances the original specification to reflect the presence of inherited elements. In other words, occurrences that are available at metamodel classes because they have been defined at a parent class are made explicit by attaching a copy of the original specification to

Algorithm 7 Determining available attribute occurrences for model objects

```
 1: function COLLECTOCCURRENCES(Attribution attribution, Object modelObject)
 2:     ▷ collect occurrences for the object's class and superclasses
 3:     objectOccurrences = new Set⟨Occurrence⟩
 4:     addOccurrences(attribution, object.class, objectOccurrences)
        return objectOccurrences
```

```
 1: function ADDOCCURRENCES(Attribution attribution, Class class, Set⟨Occurrence⟩
    objectOccurrences)
 2:     ▷ get occurrences assigned to this specific class
 3:     classOccurrences = getAssignedOccurrences(attribution, class)
 4:     ▷ examine the occurrences
 5:     for all (classOccurrence : classOccurrences) do
 6:         ▷ check if occurrence with same attribute type is already present
 7:         attributeAlreadyPresent = false
 8:         for all (objectOccurrence : objectOccurrences) do
 9:             if (classOccurrence.definition == objectOccurrence.definition then
10:                 attributeAlreadyPresent = true
11:                 break
12:         ▷ add occurrence only if its attribute type is not already represented
13:         if (not attributeAlreadyPresent) then
14:             objectOccurrences.add(classOccurrence)
15:     ▷ recursively traverse generalization hierarchy
16:     for all (superClass : class.superClasses) do
17:         addOccurrences(objectOccurrences, superClass)
```

the children. This enhanced version of the attribution can then be used to instantiate the analysis for multiple models and it can even be persisted to avoid a repeated execution of this step.

Algorithm 8 defines the function enhanceAttribution which takes the attribution that should be augmented as input. In lines [3-6], a helper map is built that associates classes with the attribute definitions which have been assigned to them via occurrences. Lines [8-10] then invoke the function inheritOccurrence for each occurrence in the attribution to attach copies to all child classes, provided they don't redefine the attribute. This method iterates over the direct subclasses [3] and retrieves the list of available attribute definitions from the precomputed helper map [5]. If the current attribute is not in this collection [7], the subclass does not provide a redefinition and a copy of the occurrence must therefore be attached to it [9]. The function then recursively invokes itself to process all children in the generalization hierarchy [12]. To avoid separate recursive calls for each branch of diamond-shaped inheritance structures, it is essential to update the helper map [11].

In its current form, the algorithm is not able to detect problematic cases such as the one shown in Figure 6.9. Instead, one of the conflicting occurrences would be inherited at random (depending on the order in which the elements are processed) which would subsequently be regarded as a redefinition. To detect this situation, the

Algorithm 8 Statically enhancing attributions with inherited attributes

```
 1: function ENHANCEATTRIBUTION(Attribution attribution)
 2:     ▷ create a map that links classes to their attached attributes
 3:     classAttributeMap = new Map⟨Class, Set⟨Attribute⟩⟩
 4:     for all (extension : attribution.extensions) do
 5:         for all (occurrence : extension.occurrences) do
 6:             classAttributeMap.put(extension.class, occurrence.definition)
 7:     ▷ iterate over all defined occurrences and inherit them to subclasses
 8:     for all (extension : attribution.extensions) do
 9:         for all (occurrence : extension.occurrences) do
10:             inheritOccurrence(classAttributeMap, extension.class, occurrence)
```

```
 1: function INHERITOCCURRENCE(Map⟨Class, Set⟨Attribute⟩⟩ classAttributeMap,
    Class class, Occurrence occurrence)
 2:     ▷ iterate over the direct subclasses
 3:     for all (subclass : class.subClasses) do
 4:         ▷ get the attributes attached to the class
 5:         subclassAttributes = classAttributeMap.get(subclass)
 6:         ▷ check if the subclass overwrites this attribute
 7:         if (not subclassAttributes.contains(occurrence.definition)) then
 8:             ▷ inherit occurrence to subclass
 9:             attachOccurrence(subclass, occurrence)
10:             ▷ update map and invoke function recursively for subclass
11:             subclassAttributes.add(occurrence.definition)
12:             inheritOccurrence(classAttributeMap, subclass, occurrence)
```

algorithm must be able to distinguish between occurrences specified in the original attribution and occurrences that were later added during the inheritance extension process. This can be achieved through a second data structure that stores the point of origin for newly created occurrences. The conditional statement in line [7] would then be able to differentiate between original specifications and newly generated elements.

Specified Attributes	
Class	**Attributes**
Class1	{Attribute1}
Class2	{Attribute1}

Added Occurrences	
Class	**Occurrences**
Class4	{Attribute1.Occurrence2}
Class3	{Attribute1.Occurrence1}

Table 6.1.: Conflict detection in the case of the example from Figure 6.9.

Table 6.1 demonstrates this method for the example presented in Figure 6.9. The first table lists the now immutable contents of classAttributeMap that reflect the attachment of the same attribute to both Class1 and Class2. Assuming the recursion first traverses the left branch of the generalization hierarchy, an entry is added to the second table that indicates the attachment of a copy of Occurrence2 at Class4.

While processing the right branch, another entry is added for Class3 which inherits Occurrence1. The problem that arises at Class4 can now be detected as the second table already contains a conflicting entry - Occurrence2 - for this class with the same attribute type as Occurrence1.

Using Data-flow Analysis to compute Inheritance

It is worth mentioning, that the inheritance of attribute occurrences can also be computed using DFA.

Algorithm 9 The attribution inherit_attributes

```
attribution inherit_attributes
    – map of available attribute occurrences
    attribute assignment availableOccurrences : Map(AttrDefinition, AttrOccurrence)
        initWith map_empty;
    – create union of locally defined occurrences and inherited occurrences
    rule eclass_availableOccurrences
        "var Map(AttrDefinition, AttrOccurrence) allAvailableOccurrences := Map{};
         for (Map attributeMap in self.superClass.availableOccurrences())
             allAvailableOccurrences := allAvailableOccurrences->include(attributeMap);
         return allAvailableOccurrences->include(self.getOccurrences())"
    – attach 'availableOccurrences' to 'EClass'
    extend EClass with {
        occurrenceOf availableOccurrences
            calculateWith eclass_availableOccurrences;
    }
}
```

The attribution shown in Algorithm 9 attaches the attribute availableOccurrences to EClass. Its result values are maps where attribute definitions are used as keys and the relevant occurrences of the respective attribute as values.

The rule eclass_availableOccurrences (which is specified in pseudocode) first queries the available occurrences from the supertypes of the respective class and aggregates them in the result map allAvailableOccurrences. It then looks up the occurrences annotated at the local class using getOccurrences() and also stores them in the map.

This process ensures that, for each attribute type, only the most specific occurrence is retained. Occurrences of the same type that are redefined at the local class will be replaced when updating the result map.

The well-formedness rule that validates whether multiple inheritance results in an ambiguous specification can also be given in the form of an attribution.

Algorithm 10 The attribution inherit_attributes_check

```
attribution inherit_attributes_check
    – checks for illegal multi-inheritance cases
    attribute constraint inheritanceCheck : error
        "illegal multi-inheritance detected";
    – check inherited occurrences
    rule eclass_inheritanceCheck
        "var Map(AttrDefinition, Set(AttrOccurrence)) conflictingOccurrences := Map{};
         for (Map attributeMap in self.superClass.availableOccurrences())
            for (KeyValuePair attributeEntry in attributeMap)
                conflictingOccurrences := conflictingOccurrences->include(attributeEntry.key,
                  conflictingOccurrences->get(attributeEntry.key)->include(attributeEntry.value));
         for (KeyValuePair attributesEntry in conflictingOccurrences)
            if (attributesEntry.value->size() > 1 and
              not self.getDefinitions()->contains(attributesEntry.key))
                return false
         return true"
    – attach 'inheritanceCheck' to 'EClass'
    extend EClass with {
        occurrenceOf inheritanceCheck
            calculateWith eclass_inheritanceCheck;
    }
}
```

The rule eclass_inheritanceCheck in the attribution in Algorithm 10 builds a map conflictingOccurrences that relates the attribute definitions that are inherited by the local class to a set of the relevant occurrences. If this set contains more than one occurrence, conflicting information is received from the supertypes. However, this only presents a problem if the attribute is not redefined at the local class anyway.

6.3.4. Running Example

We now apply the instantiation concept to the running example and visualize the structure of the resulting elements in their abstract syntax.

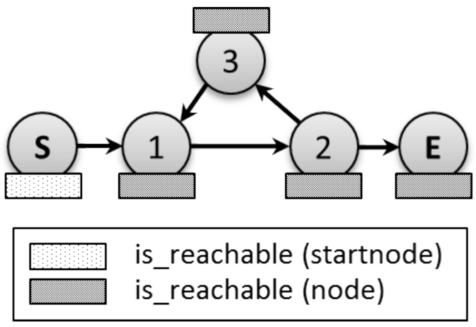


Figure 6.10.: Control-flow model with attribute instances.

Instantiating the reachability analysis for the control-flow model results in a set of instances that are attached to the graph's nodes. In Figure 6.10, the instantiated attributes are annotated as rectangles at their respective base elements. It can be seen that, although all of these instances are of the same attribute type (isReachable), they have been created by two different occurrences.

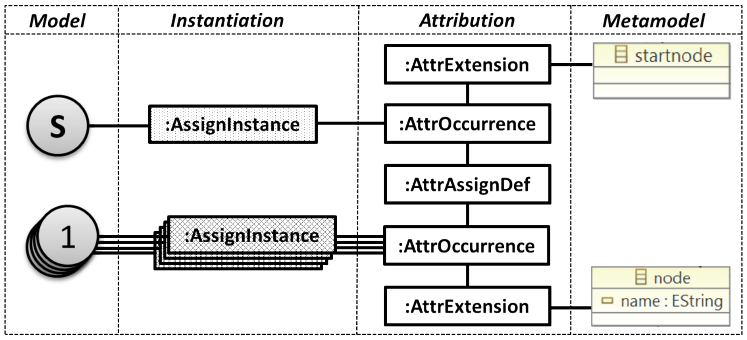


Figure 6.11.: Abstract syntax of the instantiated analysis.

The abstract syntax of the involved modeling artifacts and the connections that exist between them are shown in Figure 6.11. Because the attribute isReachable is redefined at the startnode class, the instance that has been created at node Ⓢ is connected to another occurrence than the instances at the remaining elements.

In this case, it is assumed that the attribution specification was *not* enhanced to reflect the inheritance of occurrences and instead a dynamic process such as the one provided by Algorithm 7 is used to determine the availability of attributes at model objects. If Algorithm 8 was used to extend the attribution, the endnode class would appear as a separate entity in this diagram with a copy of the occurrence that has been defined for the node class attached to it. The node Ⓔ would then be connected to this occurrence.

6.4. Rule Specification Languages

One of the design goals listed in Section 4.2.1 deals with the requirement of versatility on a technical level. In the context of the specification of data-flow equations, this refers to the ability to employ different languages to formulate these rules. The abstract syntax of attributions provides a generic interface for rules and datatype definitions and thereby offers the possibility to devise a highly versatile methodology for the inclusion of different implementation languages. This section discusses methods for adapting arbitrary executable languages for the purpose of defining data-flow rules.

In Section 6.4.1, we discuss the general properties of the rule execution interface. Concrete realizations of this technique (which can be regarded as starting points for custom adaptions) have been developed for both Java and OCL and are discussed in Section 6.4.2 and Section 6.4.3. More details about the actual implementation of these language interfaces can be found in Section 8.1.4.

6.4.1. Requirements for Execution Languages

To be able to employ different implementation languages for writing data-flow rules, a certain amount of flexibility is required on the analysis specification level. For this reason, the constructs defined by the specification language's abstract syntax realize a very generic framework for the invocation of arbitrary functions. More specifically, this pertains to the properties of rule and type specifications which themselves do not contain concrete information about the actual invocation process. Instead, attribution rules are represented as plain text that, along with an identifier for the respective execution language, is passed to the DFA solver. The solver can then invoke the rule with the help of an adapter for the specified language that is able to process the rule's contents. For this purpose, the solver module must implement an interface for each supported language which is able to interpret and execute rules written in this language. Two additional requirements must be met by these interfaces: Because the same data-flow rule may be used for the calculation of different occurrences and even different attributes, its behavior may depend on the attribute instance for which it is invoked. Therefore, the current execution context must be passed to the rule. Secondly, a method must be provided for accessing the values of other instances from inside a data-flow rule.

Because of the complex interaction patterns between the language interfaces and the DFA algorithms, we must examine several aspects of the invocation process of data-flow rules. First, we discuss the implications of the structural composition of analysis specifications and their execution semantics on the execution context of rules. Then, we investigate the process of invoking DFA rules and the related matter of requesting the values of attribute instances in more detail. Next, we outline two different approaches for defining the rules, namely the embedding of executable code inside an attribution and referencing external rule specifications.

Context of Semantic Rules

The concepts from the abstract syntax that are relevant to the execution of rules are the semantic rules themselves and the attribute datatypes. The datatype of an attribute defines the value domain for all occurrences of this attribute and subsequently for all instances that are derived from these occurrences. The return type of a rule assigned to an occurrence therefore has to match the datatype of the respective attribute.

In an attribution specification, a semantic rule is a global definition that can be used by multiple occurrences and thereby be invoked in different contexts. If the respective attribute definitions have different datatypes, this means that the corresponding rule has to return an object of the respective type depending on its execution context. This case is not very frequent as it is usually expected that a rule always computes a value of the same type[9]. However, there may be special cases, e.g. if for convenience reasons, a combined initialization rule was used to initialize attribute instances of different datatypes.

Some languages such as OCL use a strict type system that requires that all functions possess a unique return type. Because semantic rules do not necessarily ad-

[9]A more common case in which the execution context influences rule behavior would be the use of a single rule by multiple occurrences. In the running example (Figure 6.3), the `startnode` rule could have been merged with the `node` rule. Depending on the execution context's class type, the implementation can then select and carry out the corresponding behavior.

Unique contexts for the rules in the running example			
Rule	**Context Attribute**	**Context Class**	**Unique Rule Context Identifier**
boolean_false	isReachable		isReachable::boolean_false : OCLBoolean *(initialization)*
boolean_true	isReachable	startnode	isReachable::startnode : OCLBoolean *(iteration)*
node_isReachable	isReachable	node endnode	node::isReachable : OCLBoolean *(iteration)* endnode::isReachable : OCLBoolean *(iteration)*

Table 6.2.: Unique rule contexts for the running example. This information can be used to generate different method signatures that incorporate the correct datatype depending on the rule's execution context.

here to this restriction, one possibility to ensure compatibility with such languages would be to make sure that the usage pattern of rules will only generate results of a single datatype. While this restriction could easily be stated as an additional well-formedness rule, it would remove a degree of freedom from the analysis specification process.

To preserve the expressiveness of the abstract syntax, all possible contexts in which a rule might be executed have to be identified and a matching method signature has to be generated for each case accordingly. On the specification level, the relevant contexts of a rule are given by its referencing attribute definitions (*initialization rule*) and by the occurrences (*iteration rule*).

For attribution specifications, the contexts in which a rule will be applied can therefore be *uniquely* identified as follows:

- For initialization rules, the context descriptor is the name of the attribute definition combined with the name of the rule.
- For iteration rules, a unique identifier can be created by combining the name of the occurrence's target class with the attribute name.

By applying this method to the running example, we are able to derive the contexts shown in Table 6.2. Note that the endnode context stems from the static enhancement of the attribution to reflect attribute inheritance as described in the previous section. The language interface can now make use of this information to generate method signatures for the rules that include the correct return type.

If, for example, we were to define a second attribute isReachableInfo of type OCLString that returns "yes" for reachable nodes and "no" otherwise, we could modify the existing rule node_isReachable to return either a boolean or a string value, depending on the attribute context in which it has been invoked. In this case, a new context node::isReachableInfo would be generated that invokes the same rule node_isReachable but expects a return value of type OCLString instead of OCLBoolean.

Properties of Datatypes

As mentioned previously, datatypes provide the domain for the values of attribute instances. Several aspects must be considered to ensure that this concept is used correctly and consistently.

For one, there is the challenge of interoperability between attributes using different datatypes. Obviously, a semantic rule that computes a result for an attribute instance has to return a value that is compatible with the datatype of the defining attribute. This implies that the datatype must be valid for the language that is used to implement the rule. A similar situation arises if the implementation of a rule requests the value of another instance as input. In this case, the rule must be able to process the provided input parameters. Because the data-flow paths between attribute instances are not known beforehand, these restrictions cannot be checked statically. The attribution specification language does not include constructs for specifying datatype compatibility. Any implementation of a language interface therefore must include appropriate datatype conversions and runtime validation.

Because of the variety of different execution languages, the attribution technique does neither provide nor enforce a specific typing system. Nevertheless, it is possible to make use of existing typing systems by defining custom datatypes inside an attribution in the form of manual datatype specifications. These can then be used as return types by the language interfaces when generating valid method signatures for data-flow rules.

Because the rules of the running example are written in OCL, the (inbuilt) datatype OCLBoolean, which translates to OCL's Boolean type, has been assigned to the isReachable attribute. This way, if the isReachable() is invoked on a model object to access the attribute instance's value at this object, OCL's execution environment knows that the return value is of type Boolean.

Some types, for example primitive datatypes like integer or boolean, are very common and supported by almost any language. For this reason, both the abstract and the concrete syntax of the specification language provide support for some basic types of specific languages, namely Java and OCL, but do so in a very generic and unintrusive manner. This allows the user to benefit from language-specific support to a certain degree but also makes it easy to extend the existing definitions with support for additional types.

Finally, the DFA solver must be able to determine whether two values are equal to establish whether a stable fixed-point has been reached. Typed values in an execution language must therefore implement a method for equality checks.

Rule Invocation

Without going into details about specific fixed-point solving strategies, it is clear that, at some point, data-flow rules have to be invoked to calculate result values for attribute instances. Since the behavior of a rule may vary depending on its execution context, a language interface must provide a way to execute rules in the context of the respective instance.

The unique context for the invocation of a rule on the level of instantiated attributions therefore consists of the attribution-level context described above and the model element to which the instance has been assigned. Based on this information, the DFA solver is able to select the matching language interface and provide it with the data necessary for identifying the respective rule and executing it with the appropriate return type for the given context.

This process can be illustrated using the running example:

- Assuming that the solver needs to execute the initialization rule for an endnode, the attribute name isReachable is relayed to the interface for the OCL language along with the model element. Because no context class was given, the interface is able to determine that the initialization rule attached to the isReachable attribute has to be executed and its return value is of type OCLBoolean.
- Now we assume that the iteration rule for an object of the endnode type should be invoked. In this case, the rule context handed to the language interface consists of the attribute occurrence which is uniquely identified through the combination of the target class (endnode) and the attribute name (isReachable). The interface can then invoke the associated data-flow rule with the return type assigned to the occurrence's definition.

As mentioned, the behavior of a rule may depend on the context in which it is executed. Therefore each rule must be called with a set of input parameters that indicate these properties. Each language interface should therefore employ a standardized method signature that passes the execution context to the rule. This information has to consist (at least) of the target model element (corresponding to the `self` variable in OCL) and the attribute definition.

Once the execution of a rule has finished, the result value must be retrieved and potentially be converted to comply to the specified datatype. For the sake of interoperability, conversion routines can be implemented that yield a unified representation of common types. A DFA solver implemented in Java could e.g. convert an OCL result of type Boolean to a Java bool value.

When considering performance aspects of the solving process, it is also important to note that some steps of the rule invocation can be carried out statically for the whole attribution and do not need to be repeated for each model that is to be analyzed. An application of this principle can be found in the extension of the attribution to reflect a metamodel's generalization hierarchy as described in the previous section. Language interfaces may carry out various steps to prepare the execution such as a (pre)compilation of data-flow rules or the generation of lookup tables for speeding up the identification of matching rules for a given context. In this case, it is vital that rule contexts are computed statically, e.g. in the form demonstrated in Table 6.2. The distinction between the preparation step and the actual execution of a rule is presented in more detail in Section 8.1.4.

Attribute Access

Because, in contrast to traditional flow analysis approaches, data-flow paths are not known in advance, language constructs are needed that allow a semantic rule to dynamically request the values of another attribute instance as input. From the perspective of the fixed-point solving algorithm, an access made by a rule equates to a callback that prompts the solver to supply the corresponding value and to correctly handle the dependency between the requesting and the requested instance.

Generally, a callback to the DFA solver has to specify the attribute instance from which the result value should be retrieved. In practice, it is however sufficient to supply the model element and the name of the attribute to uniquely identify the corresponding instance.

Concrete methods for encoding attribute accesses of course depend on the respective execution language. Model-centric languages such as OCL may provide access to DFA attribute values by extending model objects with "virtual functions". This is demonstrated in the running example where an operation isReachable() is automatically made available at each element of type node. In languages such as Java, an accessor interface can be passed to each method for requesting instance values by providing the target element and the attribute name as parameters.

Embedded vs. Referenced Rules

The attribution metamodel (cf. Figure 6.2) specifies two fields for semantic rules. In the rule field, a string is stored that contains whatever constitutes the rule in the respective target language, e.g. Java or OCL code. The ruleType property, on the other hand, signifies which language interface should be used to execute the rule's contents. Thereby, it is not only possible to distinguish between different languages but also to provide multiple interfaces for the same language that implement different strategies to invoke rules.

In most cases, there are two different methods which can be used to encode rules: The executable code can either be directly embedded in the rule specification itself or it can be stored in an external library while the rule field only contains a reference to it. In the case of embedded code, it is possible that the statements must be compiled before they can be executed. This usually involves generating a valid expression in the target language (e.g. a Java class structure) around the rule statements and compiling it. Alternatively, if a rule specifies a reference to a library function, the respective language interface must resolve the dependency and create a method handle.

Both variants have respective advantages and disadvantages. If the actual code is stored in an external library, the attribution specification remains clear of the implementation itself which improves readability. Theoretically, it is also possible to dynamically alter rule behavior by exchanging the underlying library, thus providing different functionalities based on the same attribution. For example, we could define a single attribute for the calculation of both reachability and liveness and use two libraries that provide different implementations of the referenced methods, computing node reachability and liveness respectively. The downside to this approach is that in addition to the attribution, the DFA solver must also be supplied with the required libraries which can introduce complications on the technical level.

Correspondingly, the advantage of embedded code is that all the necessary information is directly encoded in a single specification. However, embedded code may clutter the attribution specification and has to be compiled or parsed when an attribution is loaded which has an impact on performance. Finally, editing code inside an attribution is more difficult because no language-specific editing capabilities are available. However, this problem can be solved by providing a dedicated editor for attributions (cf. Section 8.2.2).

In general, it is advisable to use embedded code for small statements and references for more sophisticated specifications.

6.4.2. OCL

The Object Constraint Language language comes in two different variants. Standard OCL supports the specification of functional statements that are free of side effects. Imperative OCL (part of the QVT model transformation framework), on the other hand, is able to modify the underlying model and introduces several imperative concepts such as conditional statements and loops. If used as a language for writing DFA rules, both variants have their respective advantages and disadvantages. The traditional form is well-suited for simple and concise statements while the imperative variant simplifies (and is sometimes necessary for) the specification of more complex functionality. However, due to the increased complexity of the language, interpreters for imperative OCL often suffer from a worse performance.

To include support for both languages, it is necessary to discuss the implications of the points presented in Section 6.4.1:

Context of Semantic Rules

The context of OCL rules can be identified using the methods presented in the last section. For each rule, constraints with appropriate method signatures can be generated for all contexts in which the rule is used. For initialization rules, OCL constraints have to be parsed without a context object while iteration rules can be defined as constraints for the respective metamodel class.

Properties of Datatypes

Both the standard and the imperative form of OCL include a rigid typing system with limited support for type casts. It is therefore necessary to assign the correct return datatype to attribute access functions (see next item).

As mentioned in the previous section, the attribution specification syntax already includes support for the most commonly used types. If additional types are required, they can be defined manually.

Since many commonly used primitive and collection types are supported by OCL, interoperability with other languages such as Java can be easily implemented by the language interface.

Rule Invocation

Iteration rules written in OCL are executed in the context of a model object which can be accessed using the keyword `self`. To execute an iteration rule, the language interface must therefore configure the interpreter with the correct context element.

To parse a set of rules in a single run, it is possible to build a library that includes all OCL rules defined in the attribution.

Attribute Access

In order to be able to request attribute instance values located at model elements, attribute access operations can be injected into OCL's execution environment (cf. Section 7.1.3). If, for example, an operation isReachable() with return type Boolean is injected in the context of the node class, it can be directly invoked on all elements of this type. The implementation of the access method then has to correctly identify the context of the requested instance and pass this information to the DFA solver. Method injection can be done either programmatically or by automatically generating a library of helper functions and parsing them alongside the original rules. To trigger the callback to the DFA solver, QVT black box methods can be used.

Because both variants of OCL provide only a limited set of functions, the injection mechanism can also be used to enrich the existing functionality with additional methods, e.g. to compute hash values for model elements as required by the SCC analysis.

Embedded vs. Referenced Rules

Since most implementations of OCL interpreters support the definition of libraries, it is possible to implement language interfaces for embedded code as well as for invoking rules defined in an external library.

6.4.3. Java

As a generic programming language, Java can be used to implement more complex logic than OCL at the cost of more verbose specifications.

Again, we examine the aspects listed in Section 6.4.1 to derive their implications on the implementation of a rule invocation interface for the Java language:

Context of Semantic Rules, Properties of Datatypes

As with OCL, contexts for Java rules can be generated using the presented method. However, this is not necessarily required since Java provides more powerful methods for type casting. Return values could therefore also be specified as generic objects with the type handling implemented in the rules themselves. For this purpose, the specification language provides a predefined type JavaObject.

Rule Invocation

The current execution context can be passed to Java rules using method parameters. For this purpose, a standard method signature has to be defined that expects the model object, the context instance's defining attribute and an accessor (see next item).

If multiple embedded Java rules are defined, a class can be dynamically constructed and compiled that aggregates these rules in a single library.

Attribute Access

To enable rules to request attribute instance values, an attribute accessor class can be implemented and passed as a parameter to the Java methods (cf. Section 7.3). This accessor has to provide a function that takes a model object and an attribute name as input, locates the corresponding attribute instance and triggers the fixed-point solver to compute and return the instance's value.

Embedded vs. Referenced Rules

Java code can be directly embedded in the attribution specification. In this case, the language interface has to wrap the code in a method using the default signature and insert the resulting methods in an empty class template. The resulting class can then be compiled and method handles for executing the rules can be extracted using Java's reflection API.

If rules in an existing class should be referenced instead, the semantic rule has to specify the classpath and the method name. Again, references to the methods can then be extracted using the reflection interface.

To simplify the initialization of attribute instances, a third interface for the Java language can be implemented that simply creates a Java object by invoking its default constructor. In this case, the semantic rule simply has to contain the fully qualified name of the Java class that should be instantiated.

6.5. Analysis Configuration and Execution

In the Sections 6.1 to 6.4, we detailed how the definition and instantiation of attribute-based flow analyses can be realized through the use of modeling technologies and how a generic interface for the invocation of semantic rules can be implemented. We will now provide a description of a flexible DFA solver architecture that is able to evaluate these specifications by employing different solving strategies to compute fixed-point results. This architecture establishes connections between an instantiated analysis (in the form of an attributed model), a specific implementation of a fixed-point solving algorithm and the language interfaces for semantic rules.

Commonly used algorithms in the DFA context rely on the propagation of data-flow results along the paths of an underlying control-flow structure. However, in the case of model-based flow analysis, the input/output dependencies between attribute instances are not made explicit in the specification. Instead, they only become visible during the execution of the semantic rules. For this reason, the traditional methods for computing flow analysis results have to be adapted, so that they are able to dynamically record and update dependency relationships between instances (*dynamic dependency discovery*). We propose several concrete algorithms that incorporate dependency discovery mechanisms and can be interchangeably used by the solver to compute fixed-point results. In addition, we will examine several optimization methods that are able to reduce the average amount of rule executions necessary to arrive at the most precise result set.

The contents of this section are structured as follows: In Section 6.5.1, we discuss the general architecture of the solver framework with respect to the requirements of our approach. Next, in Section 6.5.2 and Section 6.5.3, we present extensions of the traditional approaches to solve DFA equation systems, namely the round-robin and the worklist algorithm. We describe how these methods can be enhanced with support for dynamic dependency discovery and how they can be integrated with the solver framework. In Section 6.5.4, we introduce an approach called the *dependency-chain algorithm*. This method records relationships between instances and uses this information to dynamically build a detailed, model-based representation of the data-flow paths. In each case, we demonstrate the solving processes of the presented algorithms in the context of the control-flow graph example.

6.5.1. Solver Architecture

The process of evaluating a flow analysis requires certain input artifacts and relies on the interaction of multiple components to derive and solve the equation system. A generic framework which supports exchanging implementations of specific functionality has the benefit of providing a flexible architecture which can be easily adapted and extended. This is an important aspect since, depending on the scope of an analysis, different methods that can be used to solve DFA equation systems may have certain advantages/disadvantages over other methods. Additionally, being able to switch between multiple implementations also supports the comparison and evaluation of different approaches.

It is often the case that the efficiency of an algorithm depends on the specific properties of the current problem such as the size of the equation system and the complexity of the therein contained dependency relationships. For fixed-point algorithms, this leads to a tradeoff between the time and memory that is spent on recording and analyzing the dependencies between attribute instances and the required amount of rule executions. This tradeoff stems from the fact that algorithms may differ in the way that they manage dependencies and the scheduling of rule invocations. On the one hand, an algorithm may choose to ignore information about attribute dependency relationships and simply repeat the execution of rules until a stable fixed-point is reached. In this case, the management overhead is minimal but this comes at the cost of requiring many rule executions until the final result is available. This approach may be nevertheless be beneficial if the rules are very simple and their execution is therefore inexpensive. On the other hand, the examination of dependency relationships and a subsequent derivation of an optimal scheduling for rule executions often can reduce the number of necessary fixed-point iterations. If the execution of the rules is more time-consuming or the equation system is very large, the increased effort that goes into the management of dependencies may quickly pay off.

In the first step of the solving process, the solver must be supplied with an instantiated analysis. This artifact conforms to the attributed model which, in turn, consists of the target model and the instantiated attribution. Additionally, a subset of instances can be provided that are of specific interest to the user. The solver can then limit the computations to the requested instances and their input dependencies (*demand-driven analysis*). The actual process of calculating the results is carried out by an algorithm that implements a fixed-point solving strategy. For this purpose, the solver module must be able to access the language interfaces to invoke the data-flow rules. Additionally, the solver has to handle the callbacks that occur when the execution of a rule requests the value of another instance as input. This request must be relayed to the respective implementation of the fixed-point algorithm which is then responsible for handling this dependency and returning the requested value.

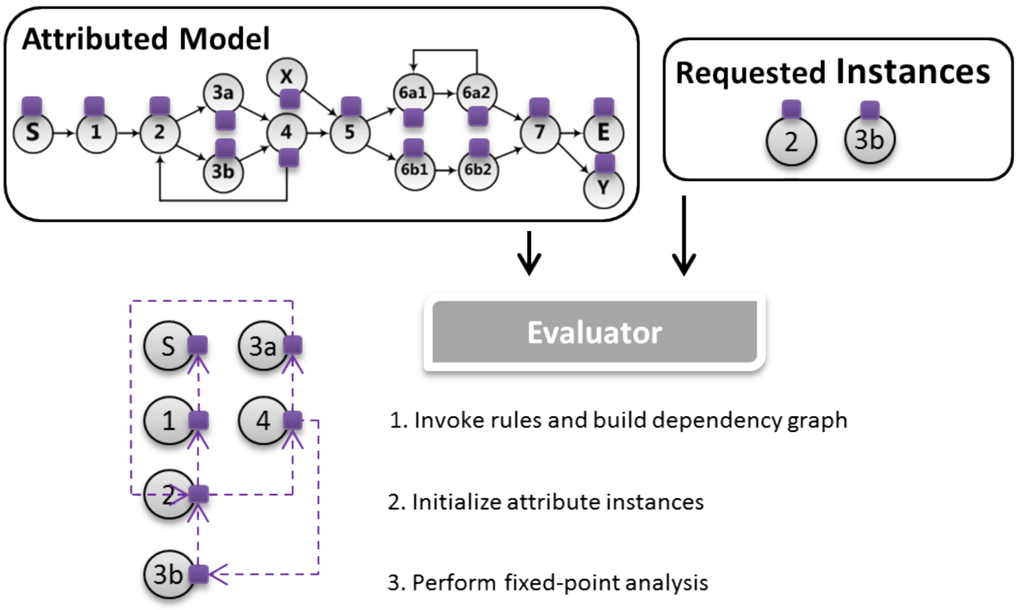

Attributed Model
S
1
2
3a
3b
X
4
5
6a1
6a2
6b1
6b2
7
E
Y
Requested Instances
2
3b
Evaluator
S
3a
1
4
2
3b
1. Invoke rules and build dependency graph
2. Initialize attribute instances
3. Perform fixed-point analysis

Figure 6.12.: Fixed-point evaluation of an attributed model.

The overall evaluation process is outlined in Figure 6.12. In this case, the attributed model consists of the control-flow graph model presented in Figure 4.12. The set of initially requested attribute instances consists of two attribute instances which are annotated at nodes ② and (3b). Performing the dependency discovery yields the dependency relationships shown in the lower left. In this case, four additional instances - Ⓢ, ①, (3a) and ④ - have been discovered that contribute to the results of the initial query. Based on information about the dependency relationships, a valid scheduling for the execution order of rules can be determined, e.g. Ⓢ→①→②→(3a)→④→(3b).

In summary, the specification of a generic architecture for DFA solvers is motivated by the goal of providing a flexible, modularized framework that divides the responsibilities between different, exchangeable components. The framework has to define interfaces for the analysis input, for the invocation of rules and for solving strategies that support the dependency discovery mechanism.

6.5.1.1. Requirements for the Solver Architecture

We will now shortly discuss several aspects that have a profound impact on the principal architecture of the solver framework:

Demand-driven Attribute Selection

In a broader sense, the input for the fixed-point algorithm consists of an attribution specification that has been instantiated for a specific model, i.e. an attributed model. However, in practice, it is sufficient to supply the solver with a set of attribute instances as the structure of the underlying (meta)model elements is not relevant for the actual result computation process.

However, the user might only be interested in the results for a specific subset of attribute instances. For example, while Algorithm 5 attaches both allPredecessors and sccID to all nodes in the model, the user may want to limit the result computation to a single instance of the attribute allPredecessors at a specific node. In this case, the input for the solver would consist of this single instance. We refer to this process as a *demand-driven* computation.

Computing the results for the set of requested instances however may require the evaluation of other instances which are not part of this initial set. In the allPredecessors example, the calculation of a specific instance will implicitly trigger the recursive computation of allPredecessors instances at preceding nodes. We therefore distinguish between results for instances which have been explicitly *requested* and results for *discovered* instances.

In summary, the input that must be supplied to the solver consists of a (sub)set of attribute instances of an attributed model while the output comprises the results for these elements as well as for dynamically discovered instances.

Dynamic Attribute Instantiation

For very large models with many attributes, the instantiation process itself may have a substantial impact on the overall performance of the analysis. It is therefore advisable to extend the demand-driven approach to the creation of attribute instances: Rather than instantiating the complete attribution, it is possible to delay the task of creating a specific instance until it is accessed by the solving algorithm.

To support a *dynamic instantiation* mechanism, the solver must be able to instruct the instantiation module to instantiate attributes on-demand. For this purpose, the solver supplies the context of the prospective instance - consisting of the attribute name and the target model element - to the instantiator. As a result of this process, the attributed model is extended with the newly created attribute instance.

Unified Rule Invocation Interface

In the previous section, we discussed the properties of language interfaces that support the specification of data-flow rules in arbitrary implementation languages. Because the rules used to compute results for attributes contained in a single attribution may be written in different languages, it is beneficial to hide this technical property behind a unified invocation interface. This interface requires two input parameters: The instance for which a rule should be invoked and an indication whether its initialization or its iteration value should be computed. In the first case, the invoker has to execute the rule assigned to the instance's definition and, in the latter case, the rule assigned to its occurrence. After the execution of the rule has finished, the invocation interface has to assign the computed result value to the respective instance.

Dynamic Dependency Discovery

If, during the execution of a rule, a request is made to retrieve the value of an attribute instance, this access is redirected to the solver. The solver then has the duty to

1. instantiate the attribute based on the provided context if this has not already happened (*dynamic instantiation*).
2. relay the request to the chosen fixed-point solving strategy (*callback*) which is then responsible for recording the dependency, determining a valid and efficient execution order and returning a preliminary result for the requested instance with which the execution of the rule can continue.

These two functions realize the feature of *dynamic dependency discovery*.

6.5.1.2. Architecture of the Solver Framework

We will now present a high-level overview of a solver architecture that realizes the features listed above. A detailed discussion of the technical aspects of this architecture can be found in Chapter 7 while Chapter 8 describes a concrete prototypical implementation.

A schematic representation of this architecture can be seen in Figure 6.13. In this diagram, the external components - the *Attributed Model* that represents the input for the analysis, the *Unified Language Interface* for executing the semantic rules and an implementation of a *Solving Strategy* - are grouped around a central *DFA Solver* module. In this design, the solver is responsible for facilitating the communication between the components involved in the solving process.

In the first step of this process, the set of requested attribute instances[10] is relayed to the Analysis Entry Point of the chosen *Solving Strategy* via the Trigger Analysis function of the solver. The entry point of the solving algorithm is then responsible for executing the rules associated with these instances.

When a request to invoke a rule is relayed to the solver module, the Invoke Rule function has to trigger the execution of the rule by supplying the correct context (cf. Section 6.4.1).

[10]While it is not necessary to instantiate the complete attribution due to the *dynamic instantiation* feature, this setup requires that at least the requested instances have been created statically.

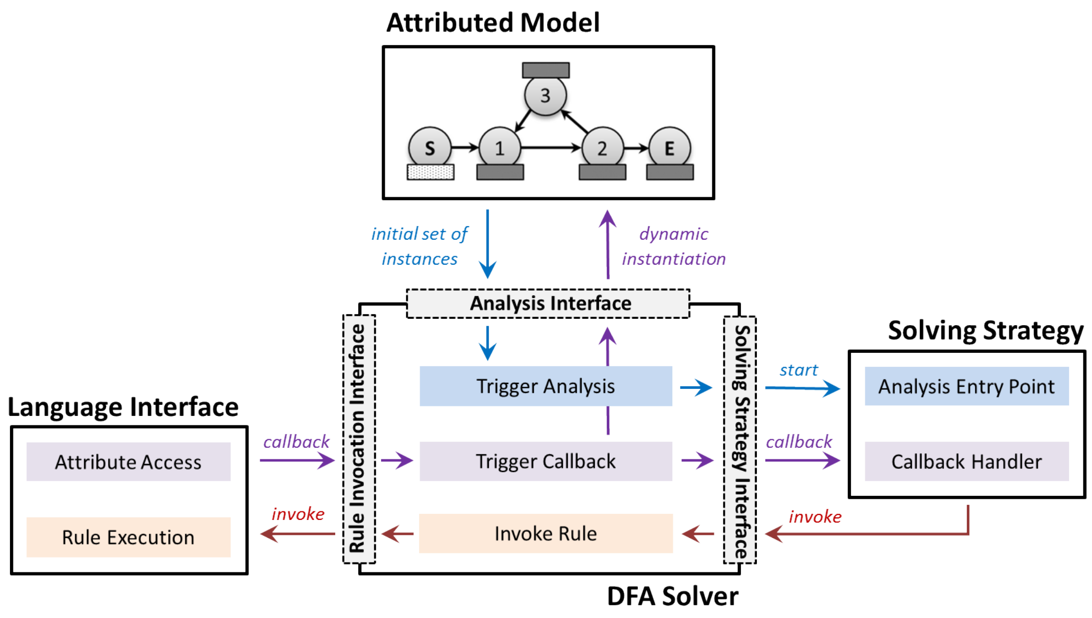


Figure 6.13.: Universal solver architecture for performing demand-driven data-flow analysis, supporting dependency-discovery and dynamic instantiation.

If, during the processing of a rule, another instance's value is requested as input, this request results in a callback that must be handled by the solver. For this purpose, the language interface passes information about the context of the requested instance to the solver's Trigger Callback function. As described above, this method now has to check if an instance already exists for the supplied context. If this is not the case, it has to be created dynamically and added to the set of discovered instances. This is accomplished through the solver's Instantiate Attribute function which performs the instantiation and updates the attributed model. The routine then has to inform the Callback Handler of the respective *Solving Strategy* that the currently executed rule requested the value of another instance as input. Once the Callback Handler has processed this dependency, the interrupted execution of the rule can continue.

Finally, when the solving strategy determines that the stable fixed-point has been reached, it has to abort the iterative computation process. The output of the analysis then consists of an updated attributed model, containing both the originally requested and the dynamically created instances and their respective result values.

6.5.1.3. Solving Strategy

Each solving strategy has to implement the following two functions:

Analysis Entry Point

The entry point of an analysis receives the set of initially requested attribute instances and is tasked with invoking their corresponding rules until a stable fixed-point has been reached.

A simple approach (such as the round-robin algorithm) may choose to evaluate all instances repeatedly by executing their respective rules and continue this process until two subsequent iterations yield the same results, in which case a fixed-point has been reached. More sophisticated approaches may start in a similar fashion, but additionally use the callback mechanism to record the data-flow dependencies between instances to derive more optimal schedulings for subsequent iterations.

Callback Handler

The callback handler is informed when, during the execution of an instance's rule, another instance's current iteration value is requested as input. The provided information consists of both the requesting and the requested instance and a flag that indicates whether the requested instance has been dynamically instantiated. This mechanism can therefore be used to construct a representation of the dependency relationships between instances.

While the concrete method used to derive an efficient scheduling for rule executions depends on the respective algorithm, there are nevertheless some issues that must be addressed by every implementation.

For one, a newly instantiated attribute may not yet have been initialized, i.e. its value is still undefined. An algorithm can therefore either choose to assign the initialization value immediately on creation or delay this step until the value is first accessed. The latter option has the potential to increase the performance of the evaluation as - under certain circumstances - it is possible to immediately assign iteration values to instances, thus eliminating the need for initialization altogether (*dynamic initialization*).

Another aspect that allows optimization under certain conditions is the *recursive lookup* of discovered instances. Generally, there are two approaches the Callback Handler can take: One option is to simply record the reported dependency and immediately return control to the requesting rule. However, this method has a significant disadvantage with respect to instance discovery: During each callback, only one instance that is not in the initial set of requested instances can be discovered. If this instance has dependencies to other hitherto undiscovered instances, a new fixed-point iteration may be necessary to unveil each of these dependencies. As a solution to this problem, the handler can schedule a recursive evaluation of discovered instances before yielding control back to the requesting rule. This way, complete chains of interdependent instances can be discovered in a single fixed-point iteration.

6.5.2. Round-Robin (Dynamic Dependency Discovery)

In the traditional round-robin method (cf. Section 2.3.3), the data-flow rules are repeatedly executed until a stable fixed-point is reached. This is possible because the order of the executions does not affect the result of the analysis. However, an inherent problem with this method is that it does not take the dependencies between attribute instances into account. Instead, it relies on a repeated evaluation of all instances which only stops after every single value remains stable.

6.5.2.1. Algorithm

Algorithm 11 implements the two functions that provide the interface to the solver architecture: AnalysisEntryPoint and CallbackHandler.

Algorithm 11 Round-robin algorithm for the DFA solver framework

```
 1: boolean unstable = true                                      ▷ marker for (un)stable results

 1: function AnalysisEntryPoint(Set⟨AttrInstance⟩ selectedInstances)
 2:     Set⟨AttrInstance⟩ iterationInstances = selectedInstances    ▷ init iteration set
 3:     while (unstable) do                                  ▷ iterate until all values are stable
 4:         unstable = false                                              ▷ reset stable marker
 5:         for all (instance : iterationInstances) do        ▷ process the iteration instance set
 6:             Object oldValue = instance.value                          ▷ remember old value
 7:             invoke(instance)                                ▷ invoke rule (triggers callback)
 8:             if (not oldValue == instance.value) then                  ▷ check if value changed
 9:                 unstable = true                                  ▷ mark iteration as unstable
10:         iterationInstances.addAll(discoveredInstances)              ▷ add discovered instances

 1: function CallbackHandler(AttrInstance requestingInstance, requestedInstance,
                             boolean dynamicallyDiscovered)
 2:     if (not requestedInstance.initialized) then                      ▷ dynamic initialization
 3:         init(requestedInstance)
 4:     if (dynamicallyDiscovered) then                ▷ recursive lookup for discovered instances
 5:         unstable = true                                   ▷ mark iteration result as unstable
 6:         invoke(requestedInstance)                              ▷ trigger recursive discovery
```

The collection selectedInstances, which is passed as a parameter to the method AnalysisEntryPoint, indicates the subset of instances from the attributed model for which results should be computed (*demand-driven evaluation*). This function also maintains a set iterationInstances that holds the instances that that are of relevance to the current fixed-point iteration. While this set is initially identical to the requested instances [2], it is later expanded to include dynamically discovered elements.

This solving strategy relies on a global variable unstable [1] that indicates whether a fixed-point has been reached. The main loop in lines [3-10] is responsible for triggering a new iteration as long as this is not the case. At the beginning of each new iteration, this marker is reset to false [4].

The processing of the instances takes place in lines [5-9]. This loop iterates over all instances that are part of iterationInstances and stores their current value in the variable oldValue [6]. It then invokes the instance's iteration rule [7] which potentially results in a callback to the method CallbackHandler. Afterwards, the old value of the attribute is compared to the new value and the iteration is marked as unstable if the values differ [8-9].

Finally, in line [10], the set iterationInstances is updated with the instances that have been discovered in the current iteration so that they will be processed in the next iteration.

As stated above, the function CallbackHandler is triggered by the solver when the execution of a rule in line [7] results in a request to another instance. For this purpose, it is supplied with three parameters: requestingInstance and requestedInstance respresent the source and the target instance of the request while dynamicallyDiscovered is a flag that indicates whether the requested instance has been dynamically created.

In its first step [2-3], this method checks whether the requested instance has already been initialized and invokes the instance's initialization rule if this is not the case (*dynamic instantiation*). This way, instances that do not possess any output dependencies do not need to be explicitly initialized.

Lines [4-6] implement the *recursive lookup* feature. If the instance has been created dynamically, it may have dependencies to other attributes which are still undiscovered. In this case, line [5] marks the current iteration as unstable as the discovered attribute must be reevaluated in the next iteration. Then, line [6] triggers the invocation of the requested instance's iteration rule. This step may again result in a callback to a hitherto undiscovered instance for which this process is repeated recursively. Because this lookup function is only carried out for newly instantiated attributes, it is guaranteed that this method will not result in an endless loop.

6.5.2.2. Running Example

We now demonstrate the application of the adapted round-robin solving strategy. As a basis for this process, we use the attributed model from Figure 4.12 which implements a predecessor analysis (cf. Algorithm 5) on a control-flow graph model. For clarity reasons, we represent an instance of the attribute allPredecessors attached to a node in the model as the node itself. For example, instead of $\langle$ ② $\triangleright$ allPredecessors $\rangle$, we simply write ②.

We assume that the set of initially requested instances that are passed to the analysis entry point consists of the nodes { ②, $\textcircled{3b}$ }. In other words, the solver is tasked to compute results for the two instances of the attribute allPredecessors which are located at the specified nodes. Additionally, the solver has to identify and evaluate instances whose values are required as input for the calculation of the two explicitly requested instances.

The steps of the solving process[11] as carried out by the round-robin algorithm are outlined in Table 6.3.

The first iteration triggers the invocation of the rules associated to the requested instances ② and $\textcircled{3b}$. Each of these calculations accesses the values of the instances located at the respective node's direct predecessors as input. Therefore, the execution of the first rule (invoke ②) results in two callbacks to the values of ① and ④.

[11]It should be noted that this is not the only order in which these steps can be executed. Depending on the sequence in which the data-flow rules are executed, the intermediate values of the fixed-point calculation may vary.

Invocation Stack	**Actions**	**Values**
Iteration 1	iterationInstances : { ②, (3b) }	
entrypoint	invoke ②	
callback ②→①	init ①, invoke ①	① : {}
callback ①→Ⓢ	init Ⓢ, invoke Ⓢ	Ⓢ : {}
return		① : {Ⓢ}
callback ②→④	init ④, invoke ④	④ : {}
callback ④→(3a)	init (3a), invoke (3a)	(3a) : {}
callback (3a)→②	init ②	② : {}
return		(3a) : {②}
callback ④→(3b)	init (3b)	(3b) : {}
return		④ : {②, (3a), (3b)}
		② : {Ⓢ, ①, ②, (3a), (3b), ④}
entrypoint	invoke (3b)	
callback (3b)→②		(3b) : {Ⓢ, ①, ②, (3a), (3b), ④}
Iteration 2	iterationInstances : { Ⓢ, ①, ②, (3a), (3b), ④ }	
entrypoint	invoke Ⓢ	Ⓢ : {}
entrypoint	invoke ①	① : {S}
entrypoint	invoke ②	② : {Ⓢ, ①, ②, (3a), (3b), ④}
entrypoint	invoke (3a)	(3a) : {Ⓢ, ①, ②, (3a), (3b), ④}
entrypoint	invoke (3b)	(3b) : {Ⓢ, ①, ②, (3a), (3b), ④}
entrypoint	invoke ④	④ : {Ⓢ, ①, ②, (3a), (3b), ④}
Iteration 3	same end results as in Iteration 2	

Table 6.3.: The steps of the round-robin solving strategy.

We assume that the request from ② to ① is processed first (callback ②→①). In the first step, the callback function initializes the requested instance ①. Then, since ① is not in the set of initially requested instances, a recursive lookup is triggered to detect any additional dependencies of ①. For this purpose, the current iteration is marked as unstable and the rule associated with the discovered node is invoked. This process leads to another callback since ① requests the value at its direct predecessor Ⓢ as input. Consequently, this instance is initialized and also subjected to a recursive lookup. However, because Ⓢ has no predecessors, the execution of the associated rule does not result in a callback. The invocation of ① is thus able to complete its computation by creating the union of the value acquired from its direct predecessor Ⓢ and the predecessor node itself: $\{\} \cup \{$Ⓢ$\} = \{$Ⓢ$\}$. Now, the request to ④ is processed (callback ②→④). Again, ④ is initialized and a recursive lookup is triggered. This process stops when (3a) requests the value of ② as ② is *not* a newly discovered instance. Once the execution of ② has finished, the same procedure is carried out for the other initially requested instance, (3b). In the final step of the first iteration, the newly discovered instances {Ⓢ, ①, (3a), ④} are added to the instance set which will be evaluated in the next iteration.

The second iteration processes the extended instance set now containing { Ⓢ, ①, ②, $\textcircled{3a}$, $\textcircled{3b}$, ④ }. Just like in the first iteration, each of the rule invocations results in a callback as the values of the respective predecessors are accessed. However, since all of the requested instances have already been initialized and discovered, the callback handler aborts without triggering any additional rule executions. We therefore have omitted these callbacks from the tabular representation. Since some instances have been updated with new values, the second iteration is also marked as unstable.

The third iteration repeats the invocations of the rules for all elements in the iteration instance set. However, during this run, no new instances are discovered and all values remain unchanged. Therefore, the results represent a stable fixed-point and no further iterations are necessary.

As a final result, the initially requested {②, $\textcircled{3b}$} and the dynamically discovered {Ⓢ, ①, $\textcircled{3a}$, ④} instances in the attributed model now possess the correct values.

It is obvious that, while this algorithm incorporates the necessary features to solve DFA equation systems in the proposed solver architecture, it is very inefficient as it neglects optimizations that can be derived from knowledge about the dependency relationships between instances. This is especially true in the last fixed-point iteration where instances are repeatedly evaluated although their input has not changed.

6.5.3. Worklist Algorithm (Dynamic Dependency Discovery)

In Section 6.5.1, we argued that for dynamic dependency discovery algorithms, there is generally a trade-off between the effort put into dependency management and the amount of required data-flow rule executions: The overhead of recording and analyzing the dependencies between attribute instances may inflict a penalty on the performance of the algorithm but this may be compensated by a considerably lower number of necessary rule invocations.

In this respect, the round-robin method presented in the last section has a very limited overhead. In fact, its permanent memory footprint solely consists of a single variable, the `unstable` marker that indicates whether a fixed-point has been reached. However, by executing rules in a random order, this algorithm neglects the fact that, if the value of a specific instance changes, only the instances that directly depend on this value as input have to be recomputed.

The traditional worklist algorithm employed in compiler construction makes use of this potential for optimization by taking into account the output dependencies resulting from the structure of the underlying control-flow graph. For this purpose, the eponymous worklist is initialized with the first node[12] in the control-flow whose value is not required as input by any other node. If the execution of the respective data-flow function results in a changed value, the output dependencies of this node, i.e. its successors in the control-flow, are added to the worklist. This process is

[12]In the case of a backward analysis, the last node is used instead.

repeated until the worklist is empty (cf. Algorithm 5).

We will now present an adaption of the traditional worklist approach for the DFA solver architecture. To conform with the requirements of this method, we must be able to determine:

1. A set of instances that can be used to initialize the worklist. The shared property of these elements is that their values are not required as input by any other instance. Since they therefore represent leaf nodes in the instance dependency graph, we will subsequently refer to them as *leaf instances*.

2. The output dependencies of any given instance. These are the instances to whom changed data-flow values are propagated.

The difficulty in the implementation of these functions stems from the fact that - in contrast to traditional data-flow analysis - the dependencies between the attribute instances are not known before the rules are actually executed. It is therefore necessary, to dynamically construct a representation of these dependencies that can then be used to schedule instances for a subsequent execution once one of their input values changes. Additionally, we have to identify the leaf instances which represent the starting point for the processing of the worklist. Again, it is also beneficial to incorporate support for recursive lookup to uncover multiple levels of dependency relationships during rule invocations, thereby reducing the number of required evaluation steps.

6.5.3.1. Algorithm

The adapted version of the worklist approach is shown in Algorithm 12.

The implementation of this algorithm maintains two global variables that are accessed by both the entry point and the callback function. The variable outputDependencies [1] stores the dynamically recorded dependency relationships between attribute instances. For each instance that represents a key in this table, the set that constitutes the respective value field encodes the output dependencies of this attribute. If it is determined that the value of an instance has changed after the execution of its data-flow rule, the output dependency set represents the instances whose values must be subsequently recomputed. The second global variable leafCandidates contains the instances that constitute the initial contents of the worklist. In theory, it would be possible to analyze the contents of the outputDependencies map to derive the set of elements which are not required as input by any other instances. However, as this map may grow very large for complex DFA equation systems, it is more efficient to compute the leaves dynamically.

Algorithm 12 Worklist algorithm for the DFA solver framework

```
1: Map⟨AttrInstance, Set⟨AttrInstance⟩⟩ outputDependencies = {}
2: Set⟨AttrInstance⟩ leafCandidates = {}        ▷ instances with no output dependencies

1: function AnalysisEntryPoint(Set⟨AttrInstance⟩ selectedInstances)
2:     leafCandidates.addAll(selectedInstances)        ▷ mark instances as leaf candidates
3:     for all (instance : selectedInstances) do        ▷ initialize output dependencies
4:         invoke(instance)        ▷ execute data-flow iteration rule
5:         if (not leafCandidates.contains(instance)) then        ▷ reset non-leaf instances
6:             init(instance)
7:     OrderedSet⟨AttrInstance⟩ worklist = new OrderedSet()        ▷ set up worklist
8:     repeat        ▷ process worklist entries
9:         for all (leafCandidate : leafCandidates) do        ▷ initialize worklist
10:            worklist.addAll(outputDependencies.get(leafCandidate))
11:        leafCandidates = {}        ▷ reset discovered leaf instances
12:        AttrInstance instance = worklist.remove(0)        ▷ pick and remove worklist entry
13:        Object oldValue = instance.value        ▷ remember old value
14:        invoke(instance)        ▷ invoke rule
15:        if (not oldValue == instance.value) then        ▷ if value changed, update worklist
16:            worklist.addAll(outputDependencies[instance])
17:    until (worklist.size == 0)

1: function CallbackHandler(AttrInstance requestingInstance, requestedInstance,
                            boolean dynamicallyDiscovered)
2:     ▷ remove requesting instance from the set of leaf candidates
3:     leafCandidates.remove(requestingInstance)
4:     if (dynamicallyDiscovered) then        ▷ recursive lookup for discovered instances
5:         leafCandidates.add(requestedInstance)        ▷ mark instance as leaf candidate
6:         invoke(requestedInstance)        ▷ trigger recursive discovery
7:         if (not leafCandidates.contains(requestedInstance)) then        ▷ reset non-leaf
8:             init(requestedInstance)
9:     ▷ add requesting instance to output dependency set of requested instance
10:    outputDependencies[requestedInstance] += requestingInstance ▷ store dependency
```

In the first step of the analysis process, the entry point registers the set of selected instances as potential leaves by adding them to leafCandidates [2]. The task of the subsequent steps in lines [3-6] consists of the computation of a preliminary representation of the data-flow dependencies and the identification of suitable leave instances. For this purpose, iteration rules are executed for the provided set of relevant instances [4]. During this process, the callback handler is responsible for recording the dependencies between the respective instance and its recursive inputs and for discarding elements that do fulfill the leaf property.

Since the worklist algorithm schedules the execution of data-flow rules based on unstable values, it is important that the preliminary results generated during this initialization step are afterwards replaced with the correct initialization values. To provide a clean and consistent starting point for the subsequent worklist processing, lines [5-6] therefore reset the values of non-leaf instances.

The main worklist loop starts in line [8]. Lines [9-10] add the output dependencies of the identified leaf instances to the worklist. It should be noted that the initialization of the worklist with the leaves themselves would be problematic. Since, by definition, their result does not depend on any external input, their repeated execution would always yield the same value and therefore lead to a premature abortion of the worklist processing.

Afterwards, the leafCandidates set is cleared. This is necessary, because the evaluation of an instance may result in the discovery of new instances if a rule's input requests are part of a conditional statement. Consequently, the preliminary state of the dependency graph that has been computed in lines [3-6] of the entry point has to be updated. This implementation ensures that output dependencies of newly discovered leaves will be processed in subsequent worklist iterations.

Line [12] then retrieves an element from the worklist. Quite similar to the round-robin approach, its current value is stored before the associated iteration rule is executed [14]. Line [15] then checks whether the instance's value has changed in which case line [16] adds its output dependencies to the worklist. The evaluation finishes once the worklist is empty.

As has been mentioned, the CallbackHandler function is responsible for recording dependencies between data-flow attributes and for identifying leaf instances. During each invocation of this method, the requesting instance is removed from the set of leaf candidates, if present [3]. The reason for this action is that an instance that requests another instance's value as input is not a leaf according to the definition and thus does not represent a valid starting point for the initialization of the worklist.

The recursive lookup feature is implemented in lines [4-8]. If the requested instance is marked as newly discovered, it is first registered as a potential leaf in the dependency graph [5]. Afterwards, in line [6], the discovered instance is evaluated. If this step triggers a callback, the element will again be removed from the set of leaf candidates by the recursive invocation of the callback handler. Otherwise, it can be concluded that the execution of the rule does not depend on values retrieved from any other instances and therefore it is a valid leaf. After the (recursive) rule invocation has finished, lines [7-8] are responsible for resetting the discovered instance to its initialization value. This process is comparable to the action taken in lines [5-6] in the entry point for the set of initially requested instances. Again, the step of resetting non-leaf instances is required to provide a clean starting point for the worklist processing.

Finally, the callback handler records the dependency. For this purpose, the entry for the requested instance in the map outputDependencies is updated with the requesting instance [10]. This step is necessary because, in addition to the discovery of instances during the processing of the worklist, it is also possible that new dependencies between already known instances are uncovered. In this case, the target instance has to be reevaluated.

In Appendix B.2, we present an alternative version of an adapted worklist algorithm. This implementation focuses on the identification of cyclic re-entry points and uses this information to schedule reevaluations of unstable nodes. For this reason, the alternative approach does not require an initialization run for the detection of leaf instances. Instead, it relies on a call stack that records the invocation order of instances during the recursive lookup phase. By checking whether a requested instance is already on the call stack, the callback handler is able to detect requests that result in cyclic dependencies and add their output dependencies to the worklist for subsequent reevaluation.

6.5.3.2. Running Example

We again demonstrate the application of the adapted version of the DFA solving strategy. For this purpose, we employ the same scenario as described in Section 6.5.2: A computation of the transitive closure of predecessors with the initial instance selection consisting of attributes at nodes ② and $\textcircled{3b}$.

Invocation Stack	**Actions**	**Output Dependencies**	**Leaf Candidates**
entrypoint	invoke ②		
callback ②→①	invoke ①		{①}
callback ①→Ⓢ	invoke Ⓢ	Ⓢ : {①}	{Ⓢ}
return		① : {②}	
callback ②→④	invoke ④		{Ⓢ, ④}
callback ④→$\textcircled{3a}$	invoke $\textcircled{3a}$		{Ⓢ, $\textcircled{3a}$}
callback $\textcircled{3a}$→②		② : {$\textcircled{3a}$}	{Ⓢ}
return		$\textcircled{3a}$: {④}	
callback ④→$\textcircled{3b}$		$\textcircled{3b}$: {④}	
return		④ : {②}	
entrypoint	invoke $\textcircled{3b}$		
callback $\textcircled{3b}$→②		② : {$\textcircled{3a}$, $\textcircled{3b}$}	

Table 6.4.: Initial computation of output dependencies in the worklist algorithm.

Table 6.4 outlines the initial phase of the presented worklist approach, i.e. the building of the output dependencies map and the computation of leaf candidates.

This process involves the invocation of the iteration rules associated with ② and $\textcircled{3b}$ and the subsequent recording of the resulting dependency relationships. In this example, we assume that ② is evaluated first which leads to a callback to ①. Since this element is not in the initial selection set, it is treated as a newly discovered instance. It is therefore marked as a leaf candidate and a recursive discovery of additional dependencies is triggered by executing its iteration rule.

This again results in a callback, this time to Ⓢ. Because ① now can be clearly ruled out as a suitable candidate for initializing the worklist, it is removed from leafCandidates. Ⓢ however, as a newly discovered instance, is registered as a potential leaf and its associated rule is executed. Since Ⓢ is the first node in the control-flow model, it has no predecessors and therefore does not request any other instance's value. Therefore, the callback handler will never be invoked with Ⓢ being the requesting instance and this element will remain in the leafCandidates set. In the last step of the callback ①→Ⓢ, instance ① is registered as an output dependency of Ⓢ, indicating that it must be reevaluated every time the value of Ⓢ changes. The recursion then folds back to the callback ②→① which records ② as being an output dependency of ①.

Since ④ is also a predecessor of ② in the control-flow graph model, the evaluation of ② will also trigger a request to this instance. During the following recursive lookup, (3a) is discovered and the dependencies between the respective instances are recorded. Any modification to the set of leaf candidates is reversed in subsequent invocations of the callback function, leaving Ⓢ as the sole starting point for the analysis.

After the execution of ② has finished, the same process is repeated for (3b). This leads to the callback (3b)→② which updates the dependency map with the final entry, the output dependency from ② to (3b).

Selected Instance	**New Instance Value**	**Worklist**
		{①}
①	{Ⓢ}	{②}
②	{Ⓢ, ①, ④}	{(3a), (3b)}
(3a)	{Ⓢ, ①, ②, ④}	{(3b), ④}
(3b)	{Ⓢ, ①, ②, ④}	{④}
④	{Ⓢ, ①, ②, (3a), (3b), ④}	{②}
②	{Ⓢ, ①, ②, (3a), (3b), ④}	{(3a), (3b)}
(3a)	{Ⓢ, ①, ②, (3a), (3b), ④}	{(3b), ④}
(3b)	{Ⓢ, ①, ②, (3a), (3b), ④}	{④}
④	{Ⓢ, ①, ②, (3a), (3b), ④}	{}

Table 6.5.: Worklist processing.

Now that an initial representation of the output dependencies between the instances has been constructed and Ⓢ has been identified as a suitable starting point, the worklist processing can start. Because Ⓢ is a leaf instance, its value (which hasn't been reset to the initialization value in contrast to non-leaf instances) can be considered to be stable. Consequently, the worklist is initialized with its output dependencies, in this case consisting of {①}. The evaluation of this instance yields a new result value {Ⓢ} which requires to schedule the respective output dependencies {(3a), (3b)} for recomputation. This is repeated until, after 9 steps, the worklist is empty, indicating that a stable fixed-point has been reached.

It is obvious that the knowledge about the dependency relationships between instances can help in deriving a reasonable execution order for the evaluation of attributes when compared to the randomized approach taken by the round-robin algorithm. However, a direct comparison with the example presented in the last section reveals that this does not necessarily lead to a reduced number of executions. In fact, for our use case, the worklist algorithm performs slightly worse than round-robin, requiring a larger amount of rule invocations as well has having a bigger memory footprint. The reason for this unexpected situation can be found in the overhead resulting from the initial derivation of dependency information. While the round-robin method is able to make use of the results that are computed during the recursive lookup step, the worklist approach requires to reset the instances to their initialization value to provide a consistent starting point for the analysis. However, this problem is mitigated by the fact that there is an upper bound for the number of rule executions whose values will be thrown away after the initialization step. For each instance, only one rule execution is necessary to determine its dependencies. Therefore, the overhead increases in a linear fashion with the number of instances. On the other hand, the intelligent scheduling of instance evaluations is able to significantly reduce the number of iterations for complex equation systems.

6.5.4. Dependency Chain Algorithm

In Sections 6.5.2 and 6.5.3, we presented adaptions of traditional approaches for solving DFA equation systems. We will now outline our own method that implements a more elaborate representation of dependency relationships that we refer to as *dependency chain*. In general, the structure of these chains mirrors the control-flow graphs that serve as a backbone for the propagation of data-flow analysis values in the traditional approach. However, there are some additional features such as the detection of cyclic structures in the dependency relationships that allow for a more thorough analysis of data-flow propagation paths.

For one, knowledge about the actual structure of dependency relationships is beneficial when parallelizing certain aspects of the solving process. This mainly pertains to the invocation of rules for which the algorithm can guarantee, based on the dependency chain's structure, that their parallel execution will not result in conflicting data accesses. A more practical advantage of this approach is that this representation supports an easy monitoring of the single steps and the overall progress of the evaluation process. This is demonstrated, for example, in Figure 8.15 which depicts a visualization of the dependency relationships and indicates the ordering for the evaluation of attribute instances that has been devised and implemented by the solving strategy. In Section 4.2.1 we motivated the use of modeling technologies to specify the structural aspects of the analysis artifacts. The same reasoning also applies to the representation of attribute instances in the form of dependency chains, for which we will consequently provide a metamodel that defines their abstract syntax.

We will now discuss several aspects that are relevant to the representation of dependency relationships. In the following list, we outline the motivations behind this endeavor:

Dependency Graph vs. Flow Graph

In traditional data-flow analysis, information is propagated along the edges of a control-flow graph. In our approach, this is however not the case. Not only is the information routed between attribute instances which form an auxiliary dependency graph that superimposes the model but the directionality of the resulting edges is reversed. Instead of specifying the target elements to which local results should be propagated, the execution of a data-flow rule yields the input dependencies of the respective instance. For this reason, we have to reverse the derived dependency relationships between attribute instances to yield the direction of the propagation of data-flow information. A directed flow edge in the dependency chain graph therefore encodes an input dependency with its source node conforming to the requested instance and the target node representing the requesting instance.

Comprehensive Propagation Paths

For model-based data-flow analysis, the dependency graph has to be constructed dynamically as dependencies only become visible during the execution of data-flow rules. Unlike the traditional approach, it is also possible that flow paths from different analyses become intertwined. For example, the computation of the attribute sccID from the attribution presented in Algorithm 5 relies on the values of allPredecessors. This means that the representation of the dependency relationships has to leverage the underlying model structure to be able to reflect the complex relationships between attribute instances.

Dynamic Extension

Because of dynamic dependency discovery, solving strategies must incorporate functions which can modify existing graphs during the evaluation phase. If additional dependencies are unveiled after the initial construction step, the respective dependency graph may need to be extended with new nodes and edges. Alternatively, it is also possible that this process results in the need to merge previously separate graphs into a single structure.

Starting Points for Fixed-Point Iterations

As has been demonstrated in Section 6.5.3, knowledge about valid starting points for the fixed-point iterations is vital to the scheduling of an optimized execution order. In a graph that reflects the input dependencies between attribute instances, the leaf nodes automatically conform to this property.

Handling of Cyclic Dependencies

While the worklist algorithm stores information about output dependencies, it does not explicitly detect cycles in the data-flow paths. Cyclic structures in the dependency graph however are important indicators for additional starting points for fixed-point iterations. For this purpose, cycles have to be identified and one of their constituents must be selected as an entry point from which any fixed-point reevaluation of the cyclic structure will be started. As a consequence, a solving strategy can then schedule a recomputation of unstable values for specific cycles.

In general, a solving strategy that employs a mechanism that is able to encode dependency relationships in the described fashion has to carry out a set of specific task to accomplish this goal. We will now give an informal description of a methodology that supports these requirements. It consists of three main evaluation stages:

Phase 0 In the first stage, an initial representation of the dependency relationships is constructed by monitoring callbacks resulting from input requests during rule executions. This also includes a detection of cyclic data-flow propagation paths. In principle, this process is quite similar to the initialization step of the adapted worklist method described in Section 6.5.3. However, instead of only using this information to record output dependencies, this method builds dependency chains containing exhaustive information about these relationships, including cyclic data-flow paths.

Phase 1 Since the dependency chain representation provides information about output dependencies, it would now be possible to employ the worklist algorithm to evaluate the attributes. However, we found that the precise knowledge about the data-flow dependency paths enables further optimizations: Starting at entry points of unstable cyclic structures, an optimized execution order for the fixed-point iterations can be derived from the graph. It is also possible to exclude stable branches from reevaluation and in certain cases postpone the execution of rules to later iterations if it is clear that their input is likely to change. Finally, a major advantage can be found in the fact that using the recorded dependencies, the calculation of rules which are independent of each other can be parallelized.

Phase 2 An inherent effect of a demand-driven approach is the lazy discovery and evaluation of dependencies if they are contained in different branches of conditional statements. This case is handled by a third stage in which newly discovered dependencies between attribute instances are incorporated into the existing dependency graphs. Afterwards, the scheduled evaluation order for attribute instances may have to be adapted to reflect the newly discovered instances and dependency relationships.

6.5.4.1. Dependency Chain Metamodel

In Section 6.3.1 we defined a metamodel that encodes the structure of attribution instantiations. The motivation behind this step was to provide a unified representation and to facilitate a consistent technical integration of all analysis artifacts. Correspondingly, we now present a model-based format that supports the management of dependencies.

For this purpose, we developed the dependency chain metamodel (DepChainMM) shown in Figure 6.14 which encodes the data-flow relationships between attribute instances and supports the management of cyclic dependencies.

The metamodel defines the following concepts:

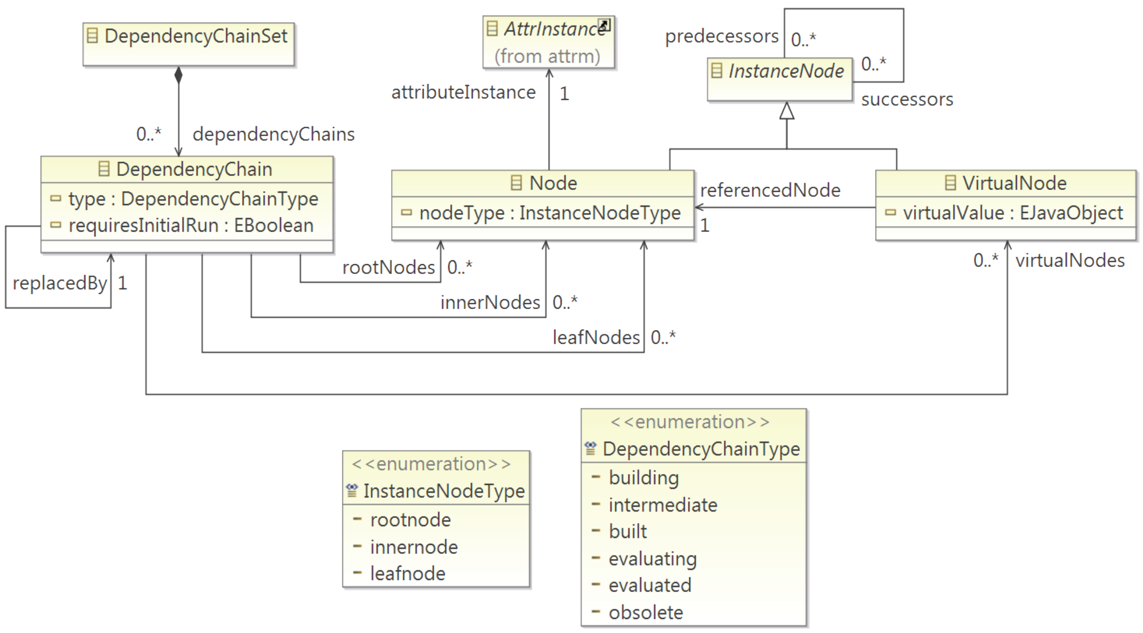


Figure 6.14.: The dependency chain metamodel (DepChainMM).

DependencyChainSet

This element acts as a container for all dependency chains that are relevant to a specific evaluation process.

DependencyChain

A dependency chain represents the data-flow relationships between attribute instances and therefore forms the basic data structure that the solving strategy operates on. Each chain conforms to a non-cyclic tree with one or more roots and leaves. Its contents are a set of interconnected instance nodes representing the attribute instances.

It is important to note that membership in a specific chain depends only on the dependency relationships of an attribute instance. As a consequence, each chain may contain instances of different attribute types and instances of the same attribute type may be part of different chains.

In contrast to DFA control-flow graphs, chains must be acyclic to enable an straightforward derivation of evaluation schedulings. The leaves of a dependency chain are either true leaf nodes which correspond to instances with no input dependencies or virtual nodes that represent entry points into cyclic dependency structures. The top-most elements of this structure are of the type root node, denoting instances with no output relationships. All other constituents that do not fulfill the listed properties are known as inner nodes.

As mentioned above, the structure of dependency chains and the types of their nodes may be altered dynamically. For example, a root node may become an inner node if output dependencies are discovered later on or two separate chains may have to be merged into a single data structure.

DependencyChainType

During the evaluation process, dependency chains are subjected to different kinds of processing. The type of a chain therefore reflects its current state.

The type building indicates that the chain is currently being constructed. After this step is finished, the state changes to intermediate as the resulting chain may still be modified, e.g. by being merged with another chain in which case the replaced chain is marked as obsolete. After the initial phase, the type of all constructed chains is set to built. The type evaluating is applied during the evaluation phase while the final results are marked as evaluated.

InstanceNode

This abstract concept defines the basic properties of dependency chain nodes. Each node may possess multiple connections to preceding and succeeding elements.

Since the relationships encode the direction of the data-flow, a node's successors represent input dependencies of the respective attribute instance. In other words, leaf nodes are nodes without successors while root nodes have no predecessors.

Node

This concept indicates the presence and the position of an attribute instance in the dependency chain graph. Each node is connected to exactly one attribute instance from the instantiation metamodel AttrM. Just like the connection between an attribute instance and its target model element, this reference is unidirectional.

As the type of a node can change dynamically, e.g. a root node becoming an inner node, the implementation of separate class types for different types of nodes would result in problems during the evaluation phase. Instead, the node type is stored in an attribute field.

InstanceNodeType

This enumeration encodes the node types root node, inner node and leaf node.

VirtualNode

Back edges to preceding nodes indicate cyclic data-flow paths. To preserve the acyclic structure of the dependency chain graph and at the same time identify entry points into cyclic components, back edges must be replaced with virtual nodes either during the chain building process or afterwards. Instead of introducing a back edge, a new virtual node has to be created that replaces the target of the dependency relationship. Because of this property, a node of this type is always a leaf of the dependency chain and denotes an entry point for the reevaluation of cycles. As once identified cycles will always continue to be part of the chain, the type of a virtual node never changes.

In essence, a virtual node is a “copy” of the target node of the cyclic dependency. This relationship is stored in the referencedNode property. After each iteration

- consisting of a bottom-up processing of the chain - the new value computed for the referenced node is propagated to its virtual copies and stored in their virtualValue field. When a new iteration n is started, the value of this field therefore represents the instance's result of iteration $n - 1$.

Iterative Dependency Chain Algorithm

As has been mentioned, it is possible to implement a dependency chain based solving strategy that relies on an iterative evaluation of data-flow attributes. While it would also be possible to evaluate dependency chains top-down in a recursive fashion, the iterative approach has two distinct advantages: In large models, data-flow propagation paths can become very long. The resulting recursive invocations may therefore exceed the capabilities of the runtime environment. Secondly, knowledge about the dependency relationships in combination with an iterative, task-based scheduler for rule executions supports a parallelization of the evaluation of unrelated data-flow results.

As underlying technique for the parallelization of data-flow computations, we employ the *thread pool* pattern. Using this methodology, each step of the evaluation process is wrapped in a task which is then submitted to a work queue. An arbitrary number of worker threads are then able to acquire the tasks from the queue and process them. The execution stops once the thread queue is empty.

The goal of *phase 0* of the evaluation process is to build the initial dependency chain structure. This function is implemented by the following steps:

1. **Parallelized Recording of Dependencies**
   The initial recording of dependencies can be implemented by a parallelized execution of the initially requested and additionally required instances. For this purpose, the selected instances are submitted to the work queue.

   Each worker thread removes an instance from the queue and invokes its associated iteration rule. The callback handler then has to initialize requested instances and relay information about the requests back to the respective thread. Consequently, after the evaluation of an instance has finished, the worker thread possesses a collection that contains all input dependencies of the invoked instance. Each of the requested instances is then itself added to the task queue.

   During this process, the worker is able to create dependency chain nodes for the executed and the subsequently requested instances. The nodes are stored in a global access map that is shared by all threads and which links attribute instances to their respective chain nodes. It is therefore important that this data structure is accessed in a thread-safe way. By querying the contents of this map, it can be ensured that each instance is scheduled for evaluation only once.

2. **Resetting the Instance Values**
   In a preparation for the iterative fixed-point computation, the instances have

to be reset to the initialization value specified for the corresponding attribute definition. This is necessary because the exploratory execution of the rules in the first step may leave the results of the attribute instances in an undefined state[13]. As initialization rules do not depend on external values, the parallelization of this step is straightforward.

3. **Assembling the Dependency Chains**
   In the last part of the algorithm's initialization phase, the interconnected dependency nodes are segregated into separate dependency chains. First, root node sets - collections of root nodes belonging to a single dependency chain - have to be identified and submitted to the task queue. For this purpose, the chain builder workers can access information generated during the recording of the dependencies. By performing a breadth-first search starting from identified leaf nodes (cf. Section 6.5.3), associated root nodes can be grouped into sets.

   The workers can then assemble the chains by classifying their nodes as root, inner and leaf nodes and replacing cyclic dependencies with virtual nodes. It is possible to speed up this process by identifying cyclic dependencies during the recording of dependencies using an algorithm that is able to detect the creation of a cycle when new edges are added. For example, the method described in [Hae+12] operates in $\mathcal{O}(m^{3/2})$ for m additions of edges.

The fixed-point evaluation in *phase 1* schedules a bottom-up processing of the dependency chain by submitting the unstable nodes to the work queue. In the first fixed-point iteration, the set of unstable nodes consists of all leaves and virtual nodes. For each subsequent iteration n, only virtual nodes whose reference value from the last iteration $it_{(n-1)}$ differs from the stored virtual value from iteration $it_{(n-2)}$ are marked as starting points for the reevaluation. If the evaluation process entered *phase 2* due to the discovery of additional instances or dependencies, an intermediate evaluation step has to be carried out. In this case, the new iteration starts with the set of newly discovered instances from which their shared predecessors have been subtracted.

The worker threads that carry out the bottom-up processing of nodes start with an execution of the associated iteration rules. Afterwards, the worker has to check for each predecessor of the evaluated node if all of its successors have either already been executed or are not scheduled for execution in the current iteration. If this is the case, the predecessor can be added to the work queue.

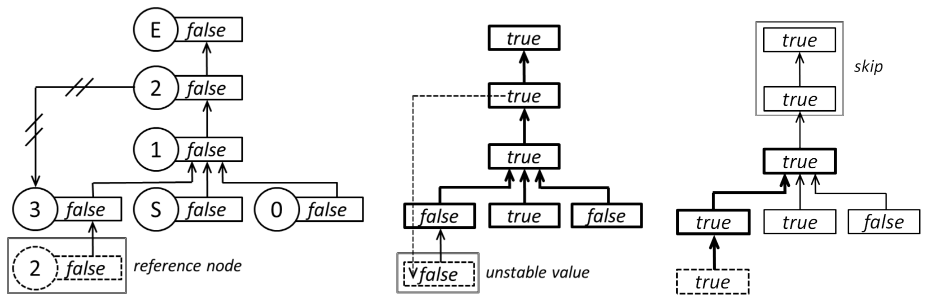
The evaluation process stops when no unstable nodes can be identified at the start of a new iteration and the task queue is empty.

In summary, the presented iterative approach is a viable solution for analyses that implement complex data-flow rules. In this scenario, the computational overhead caused by the dependency management facilities of the algorithm can be neglected in favor of the ability to make use of modern multicore processors for a parallelized evaluation process.

[13]This process resembles the initialization run of the worklist algorithm (cf. Algorithm 12).

6.5.4.2. Running Example

We will now outline the evaluation process using the dependency chain solving strategy. We apply this method to the reachability analysis (cf. Algorithm 3) and the model shown in Figure 6.10. For this purpose, we assume that a new (unreachable) node ⓪ is included as a predecessor of ①.



(a) Recorded dependencies (b) After iteration it_0 (c) Final result after it_1

Figure 6.15.: Dependency discovery and result computation [SB13].

The following description of the evaluation process is taken from [SB13]:

> In the first phase of the evaluation process, the DFA equations corresponding to the selected instances are executed. By monitoring the input requests during the rules' execution, the solver is able to construct an initial dependency graph from the recorded data-flow dependencies. The graph is then converted into an acyclic representation by identifying cyclic dependencies through a depth-first traversal strategy and replacing back edges with *virtual nodes*. Finally, all instances in are reset to their respective initialization value. This is demonstrated in Figure 6.15(a): The back edge between isReachable instances at nodes ③ and ② has been replaced by a reference node and all values have been reset to false.
>
> In the second phase, the graph is traversed repeatedly in a bottom-up fashion, starting at unstable leaf nodes. Each instance node's iteration rule can be executed once its input dependencies have been satisfied, i.e. all of its children have been either executed or do not have an unstable node in their transitive children set. Parallelization is possible if rules are executed through a working queue to which the parents of traversed nodes are added once the aforementioned condition applies. Since rules are free of side effects, it is safe to stop traversal at nodes if their execution yields the same result for an instance as in the last iteration. This avoids unnecessary recalculations of stable results. After the traversal, unstable instances at cyclic dependencies can be detected: A reference node is classified as unstable if its result from the previous iteration $it_{(n-1)}$ is different from the current iteration (it_n) value at the referenced node. As long as instances with values that differ between

iteration $it_{(n-1)}$ and it_n are identified, a new fixed-point iteration $it_{(n+1)}$ is triggered starting with the parents of the unstable reference nodes. For the first iteration it_0, all leaves are classified as unstable with the DFA initialization values representing $it_{(n-1)}$.

Figure 6.15(b) shows the result after the initial iteration with the highlighted nodes representing the executed rules. Since isReachable at the model object ② now differs from its previous value, the new result is transferred to the reference node. Its predecessor, the instance at model node ③, is scheduled as starting point for bottom-up traversal in it_1. The stable fixed-point is reached after iteration it_1, shown in Figure 6.15(c). Since the value for model object ① has not changed, the traversal can be aborted without recalculation of ② and Ⓔ.

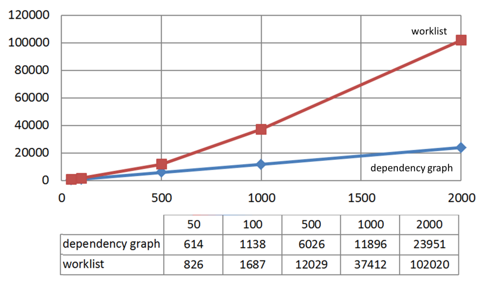
The discovery of new dependencies during the evaluation process can result in the introduction of additional nodes, the reconnection of existing nodes or the merging of previously separate graphs. To handle this case, the required modifications are postponed until after the current iteration it_n finishes. Then, an intermediate step $it_{n'}$ is carried out in which the existing graphs are extended by repeating the chain-building steps of phase 1 for the discovered attribute instances. For iteration $it_{(n+1)}$, re-evaluation is scheduled to start at the smallest set of leaf nodes that includes all newly created instances and nodes which introduced new dependencies to existing instances as parents.

6.5.4.3. Evaluation

The following paragraphs present our findings in the evaluation of the scalability of the algorithm and its performance and were originally published in [SB13]:

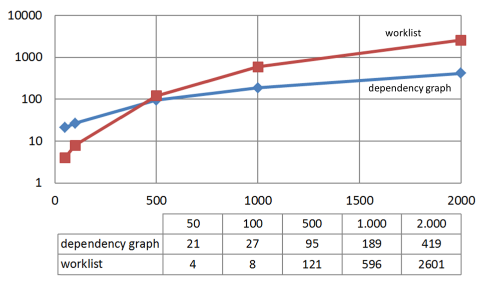
> Both the number of rule executions in relation to the amount of instances and the time for the analysis are indicators for its performance aspects. The goal is a qualitative assessment of the applicability of the approach for the analysis of large models. The evaluation employs four attributes - isReachable, allPredecessors and sccID - as well as allPredecessorsMin which calculates the dominating sets, using equivalent bitvector-based implementations of the semantic rules. To evaluate the scalability with respect to the amount of instances, five models have been generated randomly to contain 50, 100, 500, 1000 and 2000 nodes. Except the start and the final node, each node has exactly two outgoing connections to arbitrary targets. Because each attribute is calculated for each node, the number of results therefore amounts to four times the number of nodes. The computation has been carried out with the dependency chain method using a modified worklist algorithm that does not construct a dependency graph to demonstrate the unoptimized application of traditional DFA to the modeling context. The values represent the

median of 90 of 100 analysis runs (to eliminate caching issues, the first 10% have been discarded) on an Intel i7 2,20GHz computer.



	50	100	500	1000	2000
dependency graph	614	1138	6026	11896	23951
worklist	826	1687	12029	37412	102020

Figure 6.16.: Number of rule executions



	50	100	500	1.000	2.000
dependency graph	21	27	95	189	419
worklist	4	8	121	596	2601

Figure 6.17.: Time in ms (log. scale)

Figure 6.18.: Evaluation of the algorithm's performance (cf. [SB13]).

Figure 6.16 shows the total amount of rules executed in the fixed-point iterations. The time in milliseconds is pictured in Figure 6.17 using a logarithmic scale. From the results it can be deduced that while the worklist method is faster at a lower number of instances, it is soon outperformed by the dependency graph approach. This can be explained by the overhead induced by the complex data structures maintained by the graph-based algorithm. The dependency graph algorithm breaks even between 100 and 500 nodes (400-2000 instances) as the time and the amount of rule executions scales with the total number of results.

In the master thesis [Min12] our approach has been applied to detect illegal backward data dependencies in AUTOSAR (cf. Section 10.4) models. The author concludes that with an execution time of 2.4 seconds (including pre-analysis steps) for the TIMMO-2-USE breaking system use case, the *"case study shows that the analysis tool is able to cope with medium sized systems"*.

Part III.
The Model Analysis Framework

7. Architecture and Technology

The Model Analysis Framework (MAF) is an Open Source project implemented on top of Eclipse IDE [MLA10]. It provides a reference implementation of the specifications described in Part II of this thesis. During the evolution of our approach, the Model Analysis Framework not only served as a proof-of-concept demonstrator but has also been used as a research platform in order to test specific enhancements and modifications and verify the validity and practicability of the proposed solutions. The experiences gained during the implementation of the tooling therefore provided valuable feedback for the refinement of our approach.

A highly flexible and modular design along with integrated capabilities to gather statistical data enabled the evaluation of the identified objectives (cf. Section 1.2). At the same time, specific characteristics such as the performance properties of different solving algorithms and the effects of certain optimizations could be measured in the context of real-world applications. This functionality has also been used in the assessment of the case studies (cf. Chapter 10).

However, the design of this framework was not only driven by academic requirements. MAF also represents a high-quality analysis tool chain suited for usage in a productive environment. This is facilitated by an architecture that anticipates the technical integration of analysis capabilities into (existing) third-party applications. Aside from the solver component, the framework also comprises a fully featured IDE to support the complete development life cycle, consisting of analysis specification, configuration and debugging. The Model Analysis Framework has been successfully used in several research projects such as the ITEA2 project VERDE[1] and WEMUCS[2] (cf. Section 10.4). It has also been employed in several industry projects and has been the subject of student theses, including [Kra12], [Min12] and [Den13].

In the following sections, we will describe the conceptual and technical properties of the implementation: Section 7.1 outlines the principles behind the Eclipse Rich Client Platform (RCP) which provides the basis for plugin-oriented development and presents additionally required components of the Eclipse modeling ecosystem. This includes the Eclipse Modeling Framework (EMF) which realizes the underlying MDE functionality, the Eclipse OCL/QVT implementations for rule specification and the Xtext framework for the generation of model-based domain-specific language editors. The basic requirements for the framework's architectural design are listed in Section 7.2 while the proposed architecture - describing components, their layout and the integration with the Eclipse platform - is the subject of Section 7.3.

[1]Validation-driven design for component-based architectures, `http://www.itea-verde.org`

[2]Methods and tools for iterative development and optimization of software for embedded multicore systems, `http://www.multicore-tools.de`

The conceptual considerations developed in this chapter represent the foundation for the detailed overview of the components of MAF and their functions in Chapter 8.

7.1. Technological Platform

The Eclipse framework provides not only an Integrated Development Environment for software development with languages such as Java or C++, but is itself a highly configurable and extensible platform that can be customized to create IDEs for specific application domains. The choice of Eclipse as a basis for the implementation of the data-flow analysis approach was motivated by the widespread use and the maturity of the platform and its accompanying projects as well as the excellent support of MDE technologies, many of which realize OMG specifications relevant to the implementation of the DFA approach.

In Section 7.1.1, we describe the basics of the Rich Client Platform concept. The Eclipse Modeling Framework (EMF) that implements the (Essential) MOF specification is presented in Section 7.1.2. To enable the definition of DFA rules in OCL, we make use of Eclipse's OCL component which is the subject of Section 7.1.3. Finally, the Xtext framework - a workbench for the development of textual model-based domain-specific languages - is introduced in Section 7.1.4.

7.1.1. The Eclipse Rich Client Platform

The *Eclipse Foundation* has been created as a not-for-profit corporation in 2004 to host *"the Eclipse projects and helps cultivate both an open source community and an ecosystem of complementary products and services"*[3]. While many software developers are aware that Eclipse offers a sophisticated Java development environment, this represents only one - although arguably the most popular - use case of this platform [Gee05].

At the core of Eclipse lies the Equinox framework, an implementation of the OSGi specification[4] which *"provides a general-purpose, secure, and managed Java framework that supports the deployment of extensible and downloadable applications."* [OSGi] *"This small kernel [. . .] provides the environment in which plug-ins execute. Each plug-in contributes to the whole in a structured manner, may rely on services provided by another plug-in, and each may in turn provide services on which yet other plug-ins may rely. [. . .] The minimal set of plug-ins necessary to create a client application is called the Eclipse Rich Client Platform (RCP)."* [CR08]

The extent to which the Rich Client Platform can be customized through the use of the plugin system is evident from the example of the Java Development Tools (JDT), the set of plugins responsible for transforming the RCP core into a fully featured IDE for the development of Java applications. Through its flexible architecture and the high quality of its components, Eclipse has secured itself a place as a professional tool for commercial applications: *"IBM revamped its Lotus product*

[3] http://www.eclipse.org/org/

[4] Open Services Gateway Initiative, http://www.osgi.org

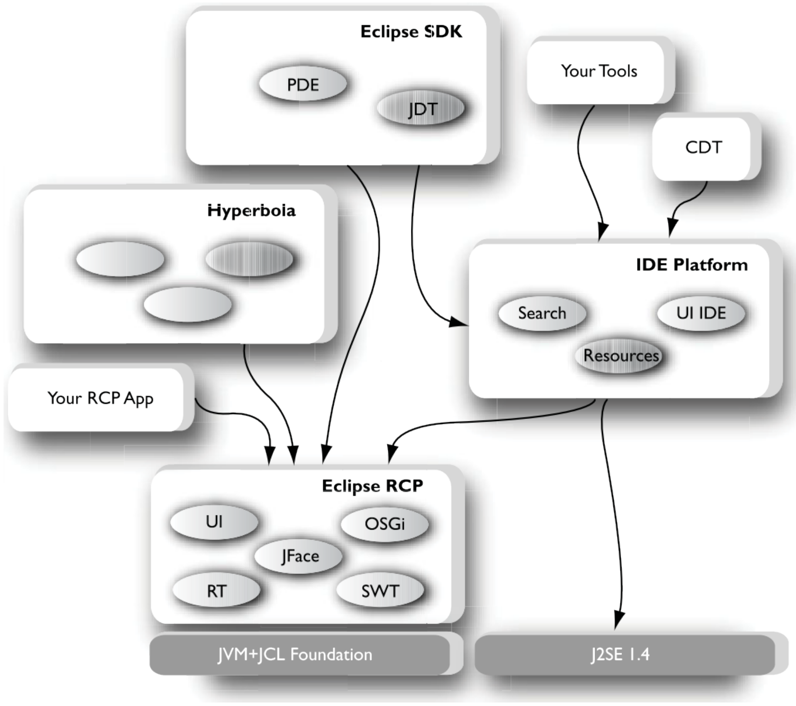


Figure 7.1.: A high-level overview of the Eclipse system architecture [MLA10].

suite to be based on RCP; NASA started using RCP for managing, modeling, and analyzing space missions; and RCP showed up unnoticed in applications in various domains. Today RCP is used as the basis of software platforms from banking and insurance to health care and geographical information systems." [MLA10]

7.1.2. The Eclipse Modeling Framework (EMF)

According to its official project page [EMFP], the Eclipse Modeling Framework is *"a modeling framework and code generation facility for building tools and other applications based on a structured data model. From a model specification described in XMI, EMF provides tools and runtime support to produce a set of Java classes for the model, along with a set of adapter classes that enable viewing and command-based editing of the model, and a basic editor."*

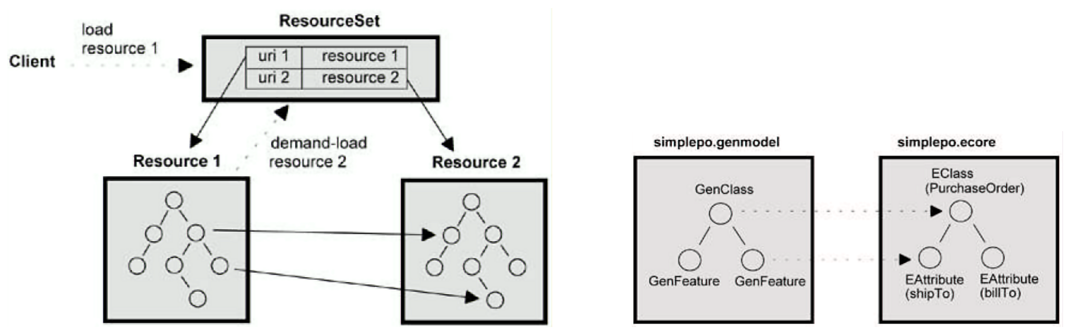
In can be observed that, in the field of academic research relating to model-driven software development, prototypic implementations are often based on EMF. Many examples for this can be found in the modeling category of Eclipse Labs[5], a project hosting platform set up by Google specifically for non-official Eclipse-related code repositories. At this time, it is home to hundreds of projects (including MAF), many of which have an academic background.

The usage of EMF is however not limited to the scientific community. It also has achieved relevance in industrial application scenarios. This is hinted at by Mike Milinkovich, the executive director of the Eclipse Foundation, in the foreword of

[5]http://code.google.com/a/eclipselabs.org/hosting/

the project's official handbook [Ste+09]: *"As a platform, EMF has transformed the modeling tools industry. Leading model tools vendors such as Borland and IBM have based their products on EMF, making a strategic decision that EMF is their core modeling framework of the future. Almost every new modeling product that I have seen over the past four years has been based on EMF."*

An important part of this success can be attributed to the wide variety of high-quality tools being developed as part of the Eclipse Modeling Project[6] which have adopted EMF as a common technical foundation for their own implementations since this framework supports building complex model-based applications through an integration of interoperable functionality. In fact, even the new release of the Eclipse platform itself (Eclipse e4[7]) makes use of the EMF facilities by internally synchronizing the state of the Eclipse workbench with a model-based representation. Some of the projects relevant to the implementation of the MAF - the Xtext DSL language workbench and the Eclipse interpreter for (imperative) OCL - are described in the following sections.



(a) EMF resource management. (b) The EMF code generator model.

Figure 7.2.: The EMF resource and generator model concepts [Ste+09].

In order to provide a better understanding of EMF and the choices that drove the design of the Model Analysis Framework, we summarize the most important aspects of the Eclipse Modeling Framework in the following paragraphs:

The Ecore model

From a technical view point, EMF's Ecore model can be considered to be a very straightforward implementation of the Essential Meta-Object Facility specification. In fact, the development of Ecore itself heavily influenced the separation of the MOF standard into a lightweight essential (EMOF) and a complete (CMOF) part. The motivation for this split was that *"with a focus on tool integration, rather than metadata repository management, Ecore avoids some of MOF's complexities, resulting in a widely applicable, optimized implementation"* [Ste+09]. Ecore therefore represents a metamodeling language that is able to reflectively define its own syntax. Within

[6] `http://www.eclipse.org/modeling/`
[7] `http://www.eclipse.org/e4/`

EMF, the terminology used for addressing constituents of the modeling language is mostly the same as in the OMG standard albeit prefixed with the letter "E"[8].

Serialization

Ecore models can be created using one of the supplied editors or alternatively be imported from a number of common formats such as Rational Rose or XML files and even annotated Java code. The default method for serializing (meta) models is the XML Metadata Interchange (XMI) format [XMI].

Implementation-wise, this is achieved through the notion of Resources as depicted in Figure 7.2(a). A Resource encapsulates a model and is able to track changes through a notification mechanism and also enables the implementation of different serialization methods such as files-based formats and databases. It is generally required that all of a model's objects must be part of a single tree structure with respect to their containment references (containment tree), i.e. no "dangling" objects are allowed. It is also possible to create interconnections between different models. If cross-references exist between elements which are part of multiple Resources, the corresponding models can be aggregated in shared ResourceSets. In this case, lazy-loading of depending artifacts is automatically managed.

Code Generation

Once an Ecore model has been created, it can be used for code generation[9]. Performing this action results in (interface and implementation) Java classes that represent the classes in the metamodel. Attributes and references are mapped to class members which can be accessed through automatically generated *getter* and *setter* methods. These methods ensure a proper handling of properties that result from the definitions in the metamodel. For example, adding an element to a containment relationship automatically removes it from its previous owner. The code generation process can be customized by annotating the model's elements with additional information which is relevant to the generation step, e.g. the output directories for the generated code. As shown in Figure 7.2(b), this configuration - referred to as *genmodel* (Generator Model) - is also a model which annotates the target metamodel[10].

If manual adjustments have to be made to the Java code, one can remove the respective Java annotations from the corresponding methods. This step creates protected areas which will not be overwritten by subsequent invocations of the code generator. Thereby, the code can be updated to reflect changes to the metamodel without overwriting manual adaptions. However, a full round-trip engineering approach in which structural changes to the code are transferred back to the model is

[8]For example EClass, EObject, EAttribute, etc.

[9]The code generation step can be customized by altering the templates - based on the Java Emitter Template (JET) technology - which are used in the Model to Text generation process.

[10]On a conceptual and on a technical level, this process is therefore very similar to the way analysis specifications extend metamodels.

currently not supported.

To ensure high quality and performance, several well-proven design patterns and best practices [Gam+95] are automatically applied during the code generation:

Interfaces
By default, interfaces and implementations are strictly separated. This is used to simulate multiple inheritance between metamodel classes, a concept which is defined in MOF but not available in Java.

Referential Integrity
References between two model elements which are navigable in both directions are always kept up-to-date on both sides. As mentioned above, containment relationships are also managed automatically.

Packages/Factories
The metamodel's EPackages are translated into package classes which provide convenient access to the Java representations of the meta elements in order to support reflection on a model level. They also hold a reference to generated EFactories which are, for example, used by Resources to create the matching Java objects during the deserialization of a model.

Adapters
The model's elements act as notifiers which publish typed change events to subscribers. This can, for example, be used to update UI controls when the model changes. In addition to implementing a publisher/subscriber architecture, the adapter pattern also acts as a mechanism to extend Java classes with new behavior without the need of subclassing to circumvent the limitation that each Java class may only have one parent. This functionality is implemented through AdapterFactories that are able to associate model elements with other objects of a given type.

Reflection/Dynamic EMF

Instead of generating code, (meta) models can also be constructed dynamically through a reflective API. In this case, default behavior is assumed for all aspects of the model. This API can also be used for static models, e.g. to access attributes and references using generic *getter* and *setter* implementations and to request meta information such as the defining EClass of an EObject instance or the EAttribute structural feature that governs the instances of corresponding class attributes.

Dynamic EMF is also automatically used as a fallback method if, during the deserialization of a model, the generated Java classes representing the metamodel's elements cannot be located, i.e. no matching EFactory can be found. This is, for example, the case when the code generation step has been omitted or the respective factory has not been registered with the runtime environment.

While dynamic EMF is generally considered to be slower than generated code due to its reflective nature, it has the distinct advantage of being able to incorporate structural changes during runtime through reflective modifications. In addition, it does not require the presence of generated code and solely relies on the Ecore/XMI file containing the metamodel specification.

Editing

Using the code generation capabilities of EMF.Edit, the derived Java equivalent of the model can be complemented with customizable editing facilities. This so-called edit code supplies the Eclipse environment with the necessary information for an automatic display of a navigable outline view based on the model's containment tree as well as showing element properties in Eclipse's property viewer. It also provides corresponding labels and icons for model elements and other information typically required to build an IDE. The edit code is also able to generate commands according to the *Command* pattern to encapsulate changes to the model such as the modification of an attribute's value or the addition/removal of elements. This enables the implementation of features such as undo/redo and drag'n'drop.

Based on this code, EMF is able to create a fully-featured editor that supports the creation and editing of models corresponding to the underlying metamodel in a tree-based representation. Other editor implementations, e.g. visual editors created with the Eclipse Graphical Editing Framework (GEF) or Xtext, profit in the same way.

7.1.3. The Eclipse OCL Implementation

The Eclipse implementation of the Object Constraint Language[11] is part of Eclipse's Model Development Tools (MDT). This project implements the OCL standard defined by the OMG using EMF technology as its technical foundation to support the validation of constraints in Ecore as well as in UML models. Internally, EMF facilities are used to encode artifacts such as the OCL standard library and parsed syntax trees of OCL expressions. Although most of the standard is supported, there are some deviations which in part stem from the fact that the official specification contains ambiguities and implicit assumptions[12][13].

The implementation features a visitor API that enables users to manually process the AST of OCL expressions and a configurable evaluation environment that represents the context in which a constraint is parsed and executed. Most importantly, it exposes the necessary functionality to load and invoke OCL queries on EMF models and to process the results.

Furthermore, the project provides IDE components for the development and execution of OCL statements. Notably, this includes Xtext editors (cf. Section 7.1.4) for the OCL language which support syntax-highlighting, content assistance and a validation of the well-formedness rules [Wil10].

[11] http://www.eclipse.org/modeling/mdt/?project=ocl
[12] http://wiki.eclipse.org/MDT/OCL/Compliance
[13] http://www.eclipse.org/modeling/mdt/ocl/docs/publications/OMG2012Mar/ADTFOCL2012.pdf

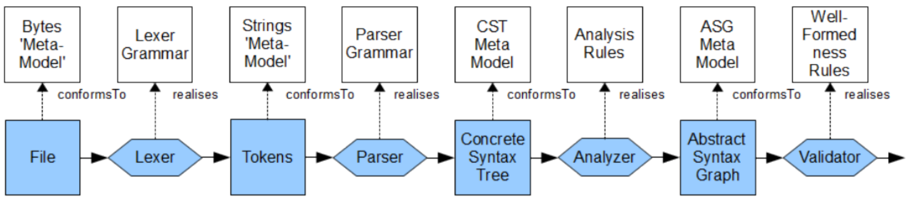


Figure 7.3.: The internal workflow of the Eclipse OCL implementation [Wil10].

Figure 7.3 depicts the steps of the evaluation process: First, the tokens of an input file are read using a lexer and parsed into a concrete syntax tree. The Analyzer component then transforms the concrete into an abstract representation, thereby simplifying the tree's structure and including semantic properties such as cross-references. Finally, the Validator checks the conformance to well-formedness rules. Both representational formats are governed by metamodels [Wil10].

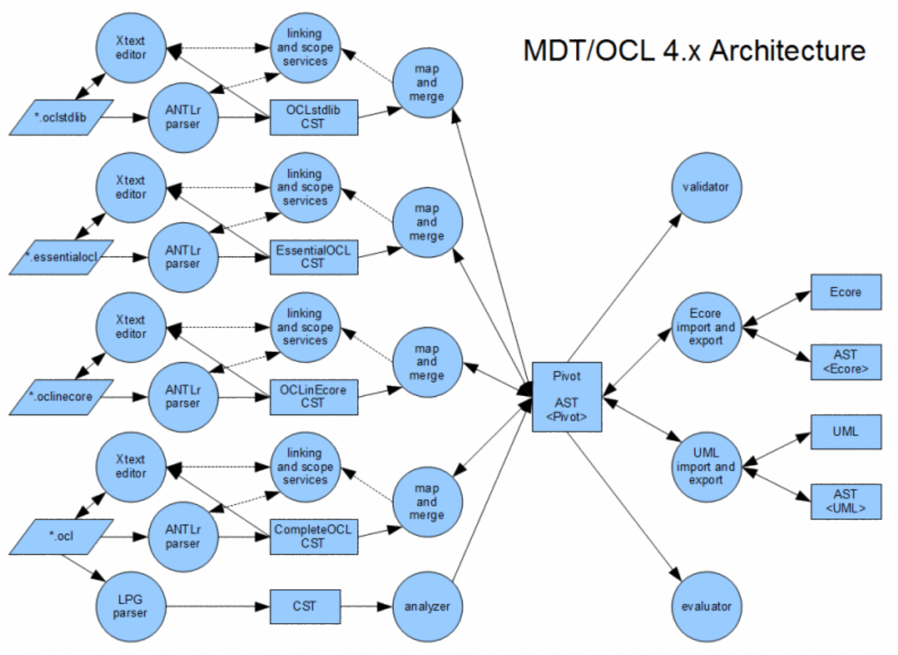


Figure 7.4.: The architecture of the Eclipse OCL project's components (`http://wiki.eclipse.org/MDT/OCL/4.X_Architecture`).

Figure 7.4 shows the architecture of the components that constitute the Eclipse OCL project. Expressions can be stored in and loaded from different formats, for example as annotations in an Ecore file (oclinecore) or as standalone queries (ocl). They are converted into a pivot representation of the AST that combines the properties of OCL queries on EMF and UML (meta) models and supports a subsequent validation with respect to OCL's well-formedness rules.

The Eclipse OCL implementation is well suited for use in the Model Analysis Framework for the following reasons:

Integration with EMF

Its tight integration with the Eclipse Modeling Framework facilitates the combination with other components of the analysis framework as MAF itself also heavily relies on the capabilities and services provided by EMF.

IDE Support
The existing editor components of the project can be reused in the implementation of a development environment for model analyses. Because of the modular properties of the Eclipse platform, it is possible to directly incorporate these editing facilities into the Model Analysis Framework's IDE.

Customizable Evaluation Environment
The evaluation of OCL statements depends on a specific context, for example the state of variables such as the `self` object. MDT OCL supports a customization of this environment without modifications to existing modeling artifacts. This simplifies the realization of the callback mechanism necessary for implementing the dependency discovery functionality required by the solving algorithms.

In the context of the Eclipse modeling facilities, the OCL project is also reused as part of the operational Query/View/Transformation implementation which belongs to Eclipse's Model-To-Model Transformation (MMT) category. This is relevant because the QVT specification [QVT] includes an extension of the purely functional OCL language with imperative concepts such as variable declarations and loops. In many cases, these constructs simplify the specification of data-flow rules. Support for black-box operations[14] again can be used to realize the callback mechanism for rules which are specified in QVT.

7.1.4. The Xtext Framework

The Xtext Language Development Framework [XTEX] is a toolkit that supports the creation of editors for text-based domain-specific languages[15]. It provides out-of-the-box support for features which are usually expected from modern development environments such as syntax highlighting, code completion, code formatting etc.

From its early days as part of the openArchitectureWare[16] distribution [EV06], it has become the main contribution of the Eclipse Textual Modeling Framework [TMF]. Xtext plays a major role in the recent trend towards the development and the usage of domain-specific languages which are specifically intended to be used by domain experts as opposed to technical experts. The goal here is to allow users without the technical knowledge required to use a general purpose programming language to participate in the development process [Gho10]. Xtext not only employs model-driven techniques but also bridges the gap between the modeling domain and

[14]Black-box operations invoke (Java) methods from inside QVT/OCL expressions.
[15]A comparison of different tooling environments for this purpose can be found in [PP08].
[16]The components of openArchitectureWare, a toolkit for model-driven software development, have since been integrated into the Eclipse Modeling Project.

textual DSLs. The popularity of this topic is reflected by the large amount of Xtext-related talks in conferences targeted at software developers[17] and its use by many open and closed source projects. According to [Beh+10] Xtext *"is used in the field of mobile devices, automotive development, embedded systems or Java enterprise software projects and game development"*. It is further stated that *"Xtext-based languages are developed for well known Open-Source projects such as Maven, Eclipse B3, the Eclipse Webtools platform or Google's Protocol Buffers and the framework is also widely used in research projects"*.

In the context of the implementation of the Model Analysis Framework, the choice of Xtext was motivated by the following facts: DSL editors created with the Xtext framework are tightly integrated with the Eclipse environment, thus enabling the realization of a consistent IDE experience. Secondly, the generated editor components use EMF as a basis for technical and conceptual integration. Finally, the framework provides the possibility to declare mappings between textual language constructs and elements in EMF metamodels and automatically synchronizes both representations.

At the core of Xtext lies a parser that implements a bi-directional mapping between the EBNF-style grammar and metamodel constructs. Parsing and interpreting a textual representation of a model therefore supports building a corresponding EMF model which in turn can be serialized to derive its textual format. The relations between the grammatical derivation rules and elements (model classes, attributes and references) in an existing metamodel are established in the grammar itself using a modified version of the Extended Backus-Naur Form. Alternatively, Xtext is also able to infer a matching metamodel from the grammar if no reference metamodel is provided.

Figure 7.5 illustrates the use of modeling technology in the Xtext workflow: The context-free grammar that constitutes the definition of the abstract syntax is represented as a Grammar Model. As mentioned, this grammar may extend an existing Ecore model, declaring textual representations for its constituents. The processing of textual inputs is carried out by a parser that is generated from the grammar specification. This results in a model-based Parse Tree representation of the input tokens. This artifact is then converted into a Semantic Model which conforms to an abstract syntax tree of the parsed language instance and at the same time represents a valid model that corresponds to an instance of the language's (Ecore) metamodel.

The overall architecture of the artifacts generated by Xtext is visualized in Figure 7.6: The editor's internal components, the parser, the linker[18] and the serializer use a common representation of the model - the XtextResource - which is a subtype of EMF's Resource class. This implementation is able to (de)serialize the model from and into its textual representation and at the same time allows access to the model's contents through EMF interfaces. Consequently, models can be processed by any EMF-based application which is built on the original metamodel.

[17] For example at EclipseCon (http://www.eclipsecon.org) which is organized by the Eclipse Foundation.

[18] The linker module manages cross-references between model elements, a feature that is not found in context-free languages.

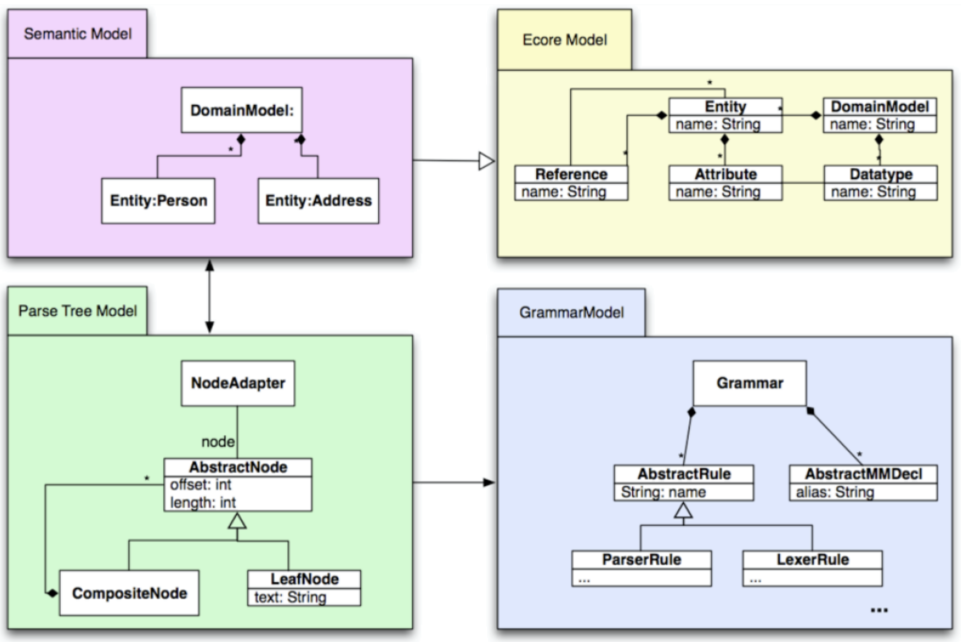


Figure 7.5.: The EMF models used by Xtext [Beh+10].

7.2. Design Goals

Development on the Model Analysis Framework began after the basic principles for the flow-based approach had been established. Originally only planned as a throw-away, proof-of-concept prototype to validate the applicability of the approach, it soon became obvious that a more carefully designed implementation could be a valuable asset for teaching and research purposes as well as for cooperation projects with industry partners.

Aside from common requirements which should be respected by every software - such as properly structured and commented code - we identified several points which are of specific interest in the context of the intended use cases:

Reuse of existing Technology
A fundamental design goal for almost any type of software can be found in the reuse of existing technology in order to avoid redundant reimplementations and improve code quality and maintainability. This goal, of course, also applies in our case. In fact, this requirement is especially relevant since the analysis approach is based on a combination of many different standards and techniques and because the tooling is intended not only to be used for research purposes but also as a high quality platform meeting the requirements of industrial application scenarios. Moreover, since MAF itself is a development platform which can be (re)used and extended by developers, its interfaces and configuration options should be exposed in a way that enables its usage in different scenarios. For example, it should be possible to include support for additional rule specification languages or extend standalone applications with the analysis capabilities provided by MAF.

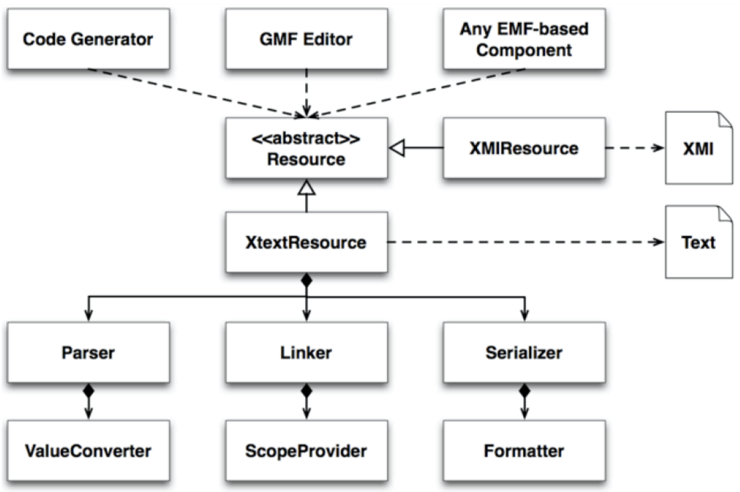


Figure 7.6.: The Xtext EMF resource as central concept of the Xtext architecture [Beh+10].

Conformance to Standards and Tools

Expanding on the *reuse* aspect, the inherent integration of standards and techniques, such as MOF or OCL, demands the employment of tools and libraries which conform to official standards rather than implementing proprietary solutions. This aspect is an important prerequisite for ensuring that the project integrates well with existing tooling. Additionally, it simplifies the integration of extensions and functions that have been developed externally such as existing model transformations between different representations. Finally, conformance to standards retains compatibility with future improvements in the involved standardization processes and the implementations thereof. From a technical viewpoint, the implementation should therefore integrate with existing platforms to provide compatibility with a wide array of existing tools.

Research Platform

Because of the co-evolvement of the theoretical concept and the technical implementation, a flexible and modular architecture was required to approach specific problems in different ways. Examples for this include the evaluation of the technical and practical applicability of multiple solving strategies for fixed-point equation systems and the integration of different rule specification languages. The research aspect is further emphasized by sophisticated methods for gathering statistical data about the analysis process, thus providing feedback for the improvement of the underlying algorithms.

Productivity Platform

Requirements for productive use differ from the properties that must be met by a research prototype. The framework has to be designed with performance and stability issues in mind. Also, the incorporation of analysis capabilities into existing applications requires an UI independent core which can be instantiated and configured through a well-defined API to control the analysis workflow.

Since the responsibility for the communication with the user lies with the third-party application, a notification mechanism must be provided. In order to support a wide range of varying technological ecosystems, it is essential that interfaces such as importers and output processors can be integrated at a later time.

Another important aspect consists of the performance with respect to time and memory constraints. The analysis core should be designed to run as a background service that is able to preload required resources such as meta-models and analysis specifications to minimize the time required for the actual execution of an analysis. Performance in the context of the evaluation algorithm can further be improved by making use of multicore processors through a parallelization of analysis tasks.

Integrated Development Environment (IDE)
Most current development platforms provide a graphical interface to simplify the access to functions and thereby allow users to focus on their respective tasks rather than spending time on issues arising from insufficient tooling support. Integrated Development Environments put an emphasis on a consistent user experience throughout large parts of the development cycle. In the context of MAF, it is therefore important that the UI-independent analysis core is complemented with graphical editors that expose as much of the capabilities of the underlying techniques as needed.

Generally, the development process of model analyses can be split into several parts: The analysis specification process itself, the configuration and resource management step and the integration of analysis capabilities into third-party applications. These issues should therefore be addressed individually.

Based on these design goals, a decision was made to employ the Eclipse Rich Client Platform and accompanying projects (the most relevant ones having been described in the previous section) from the Eclipse ecosystem as technical basis for the implementation of the Model Analysis Framework.

In more detail, the reasons for this decision are as follows:

- Eclipse itself (along with many of its related projects) are available as Open Source, often under the licensing policy of the Eclipse Public Licence (EPL)[19] which in many respects can be considered to be more friendly to industrial applications as, for example, the GNU General Public License (GPL)[20].

- Through its Equinox core (an implementation of the OSGi standard [MVA10]), Eclipse provides an advanced plugin mechanism that supports a modularized approach to software development. The services of different bundles[21] can be combined to realize more complex functionality while the dependencies that

[19]http://www.eclipse.org/legal/epl-v10.html
[20]http://www.gnu.org/licenses/gpl-3.0.html
[21]Bundle is the OSGi terminology for a plugin.

exist between these bundles (and even between different versions of the same bundle) are managed automatically by the Eclipse platform.

- Through the use of extension points, the appearance as well as the behavior and the internal functionality of the Eclipse IDE can be easily customized. The same principle also applies to third-party applications built on Eclipse RCP which can define and publish their own extension points. This is especially relevant as more and more commercial applications are developed on top of the Rich Client Platform and application development is increasingly centered on the reuse and combination of existing functional modules.

- Eclipse is currently the leading platform when it comes to the (Open Source) implementation of standards and technologies in the MDE domain, more specifically the standards released by the OMG as stated in an interview by Ed Merks[22], the project lead of the Eclipse Modeling Framework. EMF in particular is well suited to function as a common basis for the development and integration of model-based tooling.

7.3. Architecture

Based on the technological properties of the Eclipse platform (cf. Section 7.1) and the design goals that have been listed in Section 7.2, we now outline the basic architecture of the Model Analysis Framework. The stated goals result in a number of requirements that must be met by the implementation which is detailed in Chapter 8.

These requirements stem from the fact the Model Analysis Framework has to provide support for two different application scenarios: On the one hand, the framework has to support the analysis specification process by providing a suitable IDE. On the other hand, its analysis capabilities are intended to be used by domain experts rather than technical specialists. In this sense, the analysis core of MAF is not a standalone application but rather represents a customizable module that can be integrated into existing tools to enhance their abilities. For this purpose, a technological bridge between the target application and the analysis framework has to be implemented by the developer who is also responsible for supplying the necessary analysis specifications. The framework therefore has to define interfaces for customization with regard to different target domains as well as provide the functionality that enables integration with third-party products.

In the following sections, we present an architectural design that supports these application scenarios. In Section 7.3.1, we discuss the components of the analysis core while Section 7.3.2 presents the functions of the IDE. Section 7.3.3 provides an overview of the technical components.

[22] http://jaxenter.com/eclipse-modeling-framework-interview-with-ed-merks-10027.html

7.3.1. The Analysis Core

First, we will address the implications on the design of the analysis core. The core module can be split into a functional and a descriptive part: The first part comprises the analysis management and solving component while the second part consists of the language specifications (i.e. the metamodels) used for the internal and external representation of the analysis artifacts. Since MAF is not only a DFA solver but a complete framework that exposes analysis capabilities to third-party applications, it has to provide sophisticated methods for resource management as well as for analysis configuration and execution. This results in the following functional entities:

Resource Repository

The flow analysis concept requires the combination of multiple resources, some of which change more dynamically than others. Specifically, the definition of an analysis comprises the attribution itself as well as the metamodel that it extends. Usually, it is expected that these artifacts are constant in the sense that they will not change after having been loaded since they are not affected by the execution of an analysis on one or more models. Because multiple analyses may exist for the same metamodel, it is beneficial to load metamodels only once (e.g. when the framework is initialized), keeping them in memory and internally connecting them to depending attributions. The same is true for the relation between attributions and models: Because the same attribution can be applied to multiple models (or different states of the same model), the resource should not automatically be discarded once the analysis process has finished.

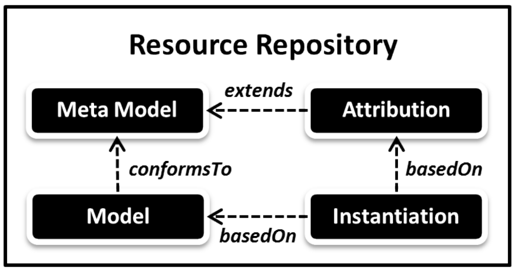


Figure 7.7.: The relationships between artifacts in the repository.

The execution of an analysis depends on two artifacts, the model and the instantiated attribution, which together form the attributed model. The concept of an Instantiation (in the context of the MAF) therefore represents a combination of three resources: The attribution, the model and - implicitly because each model and attribution resource are connected to the corresponding target language - the metamodel itself. These relationships are shown in Figure 7.7. In addition to the three input resources, this element has to hold the generated attribute instances. Again, while it would be possible to discard the resulting Instantiation resource after the execution of an analysis has finished, under certain circumstances it can be beneficial to keep it in memory. This is the case when multiple, incremental analyses should be executed subsequently -

but not immediately - on the same model. In this case, the Instantiation acts as a container that holds the already calculated attribute instance results for later access by other analyses which depend on these values.

To support these requirements, we chose to design a sophisticated internal repository management system that possesses the ability to load and hold individual resources from different sources while recording their interdependencies as depicted in Figure 7.7. This information is required during the loading and instantiation processes to locate the corresponding artifacts but it can also be used to automatically discard depending artifacts, e.g. the removal of depending attributions and models if their target metamodel is dropped from the repository. In summary, this design facilitates the semi-automatic management of the resource lifecycles which simplifies the programmatic handling and supports an optimization of the performance of the resulting application by avoiding redundant loading processes and limiting memory consumption.

Note: To distinguish between the external and internal representations, we refer to the input in its original format as artifact while the internal representation is termed resource.

Resource Adapters

An important aspect that directly relates to the notion of resource management is the need to support different input types. (Meta) models and attributions may be loaded from memory, from files using different storage formats or even databases. In some cases, it might also be necessary to apply a preprocessing step to an input artifact, e.g. a transformation that converts the input into a suitable internal representation. This requires to generate mappings which relate the resulting internal elements to their original source format so that analysis results can be correctly interpreted. Finally, the imported artifacts have to provide interfaces to the Resource Repository component described in the previous item.

We refer to the modules that implement this functionality as Resource Adapters. An adapter has to implement several interfaces that provide functionality for the import and the internal management of model artifacts. As a consequence, the analysis core can access the repository's contents via the standardized interfaces while remaining unaware of implementation-specific details that relate to the properties of different input formats.

To support customization for different technological domains, it is important that the adapter interface is exposed to developers who are then able to implement adapters for custom artifact types.

The general approach is outlined in Figure 7.8: Based on the format of the input artifact, a compatible adapter is chosen by the Resource Loader and tasked to convert the source to an internal representation which is then stored in the Resource Repository. The architectures for model and attribution Resource Adapters follow a similar pattern. Note that in this case, the resource manager is responsible for linking the internal representations of models and

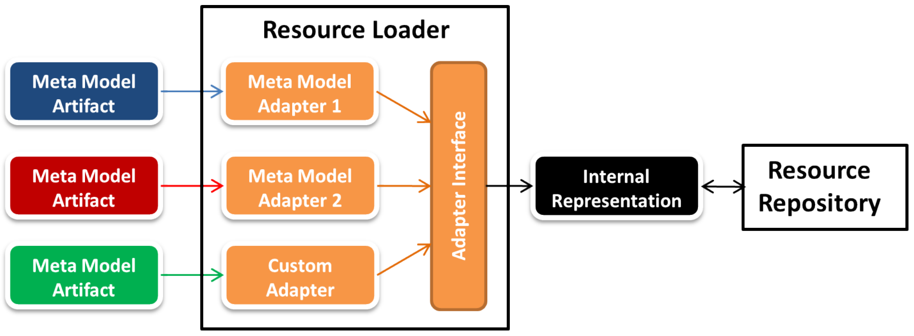


Figure 7.8.: Loading resources using resource adapters.

attributions to the respective target metamodel. The different adapter types are named Metamodel Adapters, Model Adapters and Attribution Adapters respectively.

Rule Invokers

While the declarative attribution specifications are purely model-based and thus technology-independent, the associated data-flow rules have to be written in an executable language. Because data-flow rules may be specified in different languages, this requires a certain amount of flexibility. To reduce complexity and decouple functionality, it is desirable that the fixed-point solving algorithm remains unaware of the respective rule language(s) (cf. Section 6.4). This can be achieved through a language-agnostic interface to a set of Rule Invokers which implement the required functionality for different target languages.

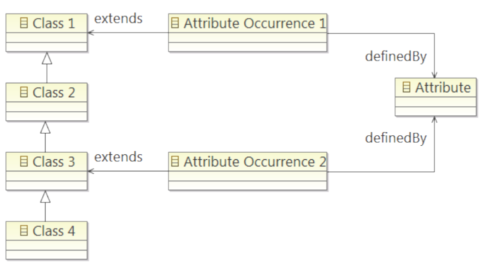
To trigger the evaluation of an attribute, the fixed-point solver hands the respective attribute instance and the information whether its initialization or iteration rule should be executed to the invocation interface. This module then has to select the appropriate Rule Invoker for the target language, task it to execute the specified rule and finally update the attribute instance with the result value.

Some implementation languages may also require to parse or compile data-flow rules before they can be invoked. The responsibility for this process also lies with the Rule Invoker for the respective language. Because data-flow rules are contained in the attributions, this step can be carried out immediately after the attribution artifact has been loaded. Since attributions are stored in the internal Resource Repository and can be used to analyze multiple models, this preparation step has to be carried out only once. For this purpose, Attribution Adapters can implement a dedicated preprocessing step that categorizes all contained data-flow rules according to their language type and task the respective invokers to convert them into an executable format.

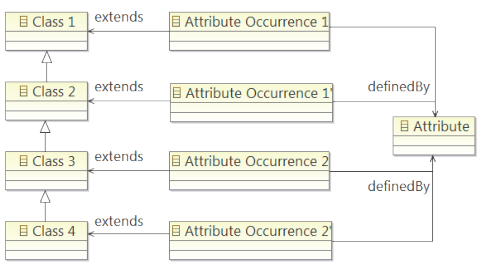
As is the case with Resource Adapters, developers that intend to incorporate analysis functionality in their own products should be able to extend the provided functionality with their own implementations. In the context of the invocation interface, support for additional rule execution languages must be included. Therefore, an extension mechanism and suitable interfaces must be provided for registering custom Rule Invokers.

Attribute Instantiation
As mentioned, instantiating an analysis requires the combination of three types of elements: A metamodel, an attribution that encodes the analysis and the target model. The instantiation process involves the traversal of the model's elements and the creation of attribute instances according to the declarative attribution specification.

In this context, the relevant properties of the metamodel are the generalization relationships between its classes. These have to be taken into account because the semantics for the instantiation step demand that data-flow attributes defined at a superclass are inherited to subclasses while respecting redefinitions at inheriting classes. However, depending on the amount and the structural properties of generalization relationships, this can have a significant impact on the performance as the generalization hierarchy must be traversed during the creation of each attribute instance. To alleviate this problem, attribute inheritance can be handled on the meta level, thereby greatly reducing the required effort, especially if the same analysis is reused multiple times. For this purpose, inheritance relationships in the metamodel are analyzed and the attribution is extended accordingly (cf. Section 6.3.3): Attribute occurrences are copied and annotated at all classes at which they are available according to the defined generalization semantics. Consequently, instantiating an attribute occurrence requires only to search for all model elements with are of the occurrence's target class type. Just as with the preprocessing of data-flow rules described in the previous item, this step can be carried out statically by the Attribution Adapter immediately after the artifact has been loaded.



(a) Original attribution



(b) Extended attribution

Figure 7.9.: Extending attributions to reflect generalization semantics.

This process is illustrated in Figure 7.9: Figure 7.9(a) shows an attribution that defines two attribute occurrences for four classes which form a single generalization hierarchy. The situation after the preprocessing step is depicted in Figure 7.9(b): The attribution has been extended with two new occurrences that reflect the inheritance semantics. Using the extended form of the attribution, instantiable attributes can be identified by checking the model element's concrete class type. It is evident that the performance gain is more significant if the statically enhanced attribution is reused to instantiate attributes for multiple models.

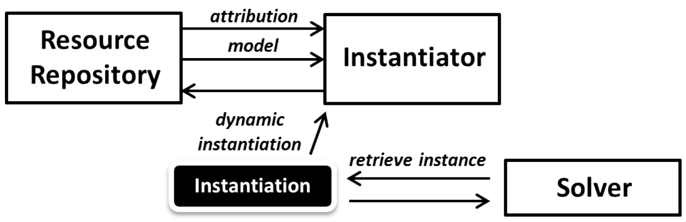


Figure 7.10.: Dynamic instantiation of attributes.

Because the evaluation of attributes follows a demand-driven pattern and consequently, not all attributes may have to be instantiated for a specific analysis, the performance can be further improved if the instantiation is carried out dynamically (cf. Section 6.3.3): While a single attribution may specify many attributes, not all of them may be of interest at a given time. Because the proposed fixed-point algorithms support dynamic dependency discovery, instantiation can be postponed until an instance is actively requested by the solver (cf. Section 6.5.1). If the same attributed model is subjected to multiple analyses, the instantiation container starts out with the instances requested by the initial analysis and "grows" subsequently to accommodate the instances created during later evaluations.

This approach allows for the instantiation containers - representing the attributed models - to be empty on creation. Only after a request to retrieve the value of a specific attribute instance is made, this instance is actually created. For this purpose, the instantiation container must provide an interface that allows other components to request single attribute instances. More specifically, the Strategy Executor as well as the Attribute Accessor must be able to instantiate attributes in advance and during the execution of an Evaluation Strategy (see below).

Figure 7.10 provides a simplified overview of this process: Using an attribution and a model as input, an attributed model in the form of an Instantiation resource is created. Attributes are then instantiated as needed by the components involved in the solving process.

Using this architecture, the choice between a static and a dynamic instantiation strategy does not affect other parts of the framework as this process is completely transparent.

Attribute Accessor

In contrast to traditional data-flow analysis, data-flow rules for attributed models cannot automatically be supplied with input parameters due to the lack of an underlying dependency graph. In Section 6.5.1, we described how the conventional algorithms can be modified to circumvent this problem. Effectively, requests for input values are implemented by the rules themselves while the solving algorithm has to monitor and record the resulting dependency relationships to update unstable intermediate results. For this purpose, input requests have to be relayed back to the solver rather than triggering the recursive execution of the respective data-flow rule.

The component that is responsible for implementing this feature is the Attribute Accessor. This module is called by the Rule Invokers any time a data-flow rule makes an input request (e.g. return the instance value of attribute X at model element Y). The Attribute Accessor then has to retrieve the corresponding instance from the Attribute Instantiation (triggering a dynamic instantiation if the attribute has not yet been instantiated) and notify the solver about the resulting dependency relationship via its callback method. Because the accessor is invoked by data-flow rules, each implementation language has to be supplied with a corresponding interface (cf. Section 6.4.2 and Section 6.4.3).

Strategy Executor / Evaluation Strategy

Due to the potentially large size of models and attributions, it can be beneficial to restrict the evaluation to a relevant subset of these resources. In some cases, it might even be necessary to impose an order on the evaluation of attributes to ensure the analysis yields the correct results. The incremental evaluation of specific attributes is supported by the lifecycle concept for resources: Instantiations represent containers for computed results, providing the input for later evaluations of depending attributes.

The selection of attribute instances which should be computed at a given time is encoded in Evaluation Strategies. A strategy consists of a sequence of Evaluation Directives which come in two different types, Evaluation Targets and Evaluation Macros. An Evaluation Target instructs the Data-flow Solver to evaluate a specified subset of attribute instances. Evaluation Macros on the other hand, execute non-DFA functions during the automated evaluation process.

A strategy is processed by the Strategy Executor component. In the first step, the attribute instances requested by the Evaluation Directives have to be instantiated statically[23]. Then, the directives are processed one by one. The execution of Evaluation Targets involves the invocation of the Data-flow Solver while user-defined Evaluation Macros are executed by calling their respective implementing function.

Data-flow Solver

The described architecture decouples the actual solver from the resource management system as well as from the instantiation process, the rule invocation service and the configuration of complex evaluation strategies. This approach has two advantages: Without the overhead of unrelated management functionality, the solver module can focus on the implementation of the actual

[23]Dynamic instantiation is limited to attributes which are detected during the dependency discovery steps of the solving algorithms.

fixed-point algorithms. Additionally, any realization of a specific fixed-point solving strategy can be easily substituted.

The Strategy Executor supplies the respective solver with two parameters: A reference to an Attribute Instantiation (possibly already containing partial results from earlier analyses) and the subset of therein defined attribute instances for which values should now be computed. The solver is then responsible for invoking the initialization and/or iteration rules associated with the requested instances in a suitable order until the values of the requested and additionally discovered instances converge in a stable fixed-point.

If the invocation of an iteration rule results in a request to another instance, the Accessor module relays this information to the solver and, if necessary, takes care of the dynamic instantiation of newly discovered instances. As a consequence, each concrete implementation of a solver has to realize two methods: The Analysis Entry Point which is called by the Strategy Executor to compute an analysis encoded in an Evaluation Target and a Callback Handler function that is invoked by the Accessor each time a data-flow rule requests an instance's value.

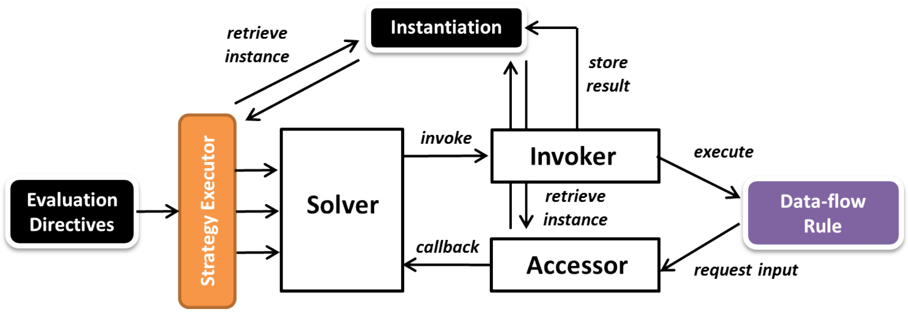


Figure 7.11.: Interaction between the modules during the evaluation.

The interactions between the different parts of the framework during the evaluation process are depicted in Figure 7.11: The Strategy Executor first instantiates the attributes requested by the Evaluation Directives and then proceeds to execute the directives one by one. For each Evaluation Target, it invokes the Data-flow Solver with the Instantiation container and the set of requested instances. The solver then executes the instances' initialization/iteration rules via the Invoker interface. Input requests made by data-flow rules are first processed by the Accessor which retrieves or creates the respective attribute from the Instantiation before relaying the request back to the solver. Depending on the chosen algorithm, this may result in additional invocations of data-flow rules. After the execution of the original rule has finished, the newly computed value is stored in the attributed model.

Result Processor

Once the execution of a strategy has been completed, results must be handed back to the main application. This happens in an asynchronous way so as not to block the caller and to enable concurrent evaluations runs. For this purpose, MAF implements a Result Processor interface. Processors can be automatically loaded during the framework's initialization phase or registered later on.

The information which processor(s) are responsible for handling the results of which Evaluation Strategies can either be specified manually when invoking the strategy or it can be specified via an Analysis Configuration (see below).

Once the execution of an Evaluation Strategy has finished, the configured processors are supplied with the computed results and the mappings of the Resource Adapters that connect the internal representations to the original artifacts. The processors can then refine the raw results before displaying them in an appropriate fashion. Unsophisticated processor implementations may simply write the output to the console or to a file.

Debugger

To support analysis development processes, the framework has to provide debugging capabilities: If running in debug mode, information about the internal actions of the components should be gathered and reported. Since the analysis core itself does not include a graphical interface, this functionality is also implemented using a notification mechanism. For this purpose, in addition to the Result Processors which are invoked after an analysis has finished, three additional types of notifiers - Repository Loggers, Instantiation Loggers and Evaluation Loggers - can be implemented that advertise information about the internal status of the framework and the steps of the evaluation process. Their respective listener functions are invoked automatically on changes to the contents of the Resource Repository and on any action carried out by an Instantiation and the Data-flow Solver module respectively. This functionality connects a graphical debugging interface to any running instance of the analysis framework or even to a dedicated debugger IDE.

A second feature of the debugger is the automatic gathering of statistical information about different tasks such as instantiating attributes, executing rules, etc. This information can be used to optimize analyses by identifying bottle necks such as data-flow rules that take a disproportionally long time to execute. In the context of this thesis, this functionality has also been vital in the evaluation and the improvement of the implemented algorithms and the case studies.

Finally, a specialized test mode can be realized which is able to execute an analysis multiple times and validate the result after each iteration. If analyses rely on complex data structures which cannot be arranged in a semilattice, is generally not guaranteed that the evaluation converges in a unique fixed-point. The test mode is a pragmatic solution to this problem as it can detect incorrect analysis specifications that yield indeterministic results. This function also helps in the detection of illegal call-by-reference modifications made by incorrectly implemented data-flow rules. These occur when a rule unexpectedly (i.e. not under the guidance of the solver) modifies an attribute value.

Analysis Configuration

The execution of an analysis requires a substantial amount of information that is not related to the actual fixed-point computation. This includes references to the input artifacts and their types, the Evaluation Directives that control the instantiation and the evaluation processes and the Result Processors. The modules of the framework may also require additional parameters to configure their behavior. While it is possible to achieve this programmatically through the framework's API, a more convenient way consists of a configuration file - called a Project Set - that specifies all of this information. Using this method, only three inputs are required to set up and execute an analysis: The Project Set containing the configuration details, the target model and the name of an Evaluation Strategy.

We will now investigate whether the proposed architecture is suitable for the relevant application scenarios. More specifically, we will argue that the framework is able to support the developer during the analysis specification process while also providing the basis for the incorporation of analysis capabilities into third-party applications:

- The core functionality for analysis configuration and execution should be focused in a light-weight library. This is achieved through the separation into a UI-independent analysis core library (that can be incorporated into existing applications) and an additional IDE with extended visualization and debugging capabilities that are usually not shipped with the final product. Furthermore, the design includes only few dependencies to other libraries which in part are optional (e.g. support for OCL/QVT) and can be excluded if they are not used to reduce the size of the application.

- To support different modeling technologies and input formats, the internal functions have to be decoupled from the technological ecosystem in which the framework is executed. This is made possible through the Resource Adapter and Result Processor concepts. By providing custom implementations of the corresponding interfaces, developers can realize compatibility with the formats, tools and standards of the target environment.

- Finally, it is possible to run the framework as a (background) service. This is necessary if the target application is not based on Java/Eclipse, in which case the analysis functions have to be addressed in a technology-independent way. Since the analysis core provides a well-defined API and a sophisticated internal resource management system, this can easily be achieved, for example by accessing the respective API calls via a network protocol.

7.3.2. The Integrated Development Environment

The IDE that supports the development and debugging of analyses consists of two parts: An editor for attribution specifications and a project editor with integrated debugging capabilities:

Analysis Editor

In Section 6.2, we provided a textual syntax for the specification of attributions. With the Xtext language workbench, the development of a corresponding DSL editor is straightforward: Based on a grammar which assigns textual representations to the elements in the attribution metamodel, an editor plugin can be generated automatically. To support the annotation of target metamodels, the editor must then be extended with a function for the import of existing Ecore files so that their elements can be cross-referenced by the attribution model's elements.

The standalone Xtext editor has to be integrated with other tooling to support the simultaneous development of attributions and data-flow rules. By making use of Eclipse's composing facilities, the editor for the attribution DSL can easily be integrated with editors for rule languages such as Java, OCL and QVT. This combination yields a sophisticated Analysis Editor that supports the relevant aspects of the analysis specification process.

Project Set Editor

A separate tooling environment is needed for the configuration and the debugging of analyses. A domain-specific language for the specification of Project Sets has to support the declaration of input artifacts and the respective Resource Adapters. It must also provide the possibility to define one or more Evaluation Strategies alongside the necessary parameters that govern the instantiation and the solving processes. If the Project Set format is specified through a metamodel, the Xtext framework can again be used to generate a corresponding DSL editor.

To support the development lifecycle, the resulting Project Set Editor should be extended to include an instance of the analysis core. This way, complete analysis configurations can be loaded and tested while providing direct feedback to the user about the validity of the specifications and supporting an inspection of the current state of the framework, e.g. the contents of the Resource Repository.

7.3.3. The Component Stack

The stack that comprises the Model Analysis Framework is outlined in Figure 7.12. The Eclipse Rich Client Platform with its OSGi-based plugin architecture represents the common technological foundation for the integration of all functional modules[24].

In a similar fashion, the Eclipse Modeling Framework provides the underlying facilities and a unified API for modeling-related functionality which is reused by higher-level components for the definition, interpretation and manipulation of structured information. This applies, for example, to the Xtext and the OCL projects which are based on EMF technology and depend on (meta) models as input and

[24]If MAF runs in standalone mode, the OSGi services are not available. Therefore, in this case some functions that are provided for convenience reasons - e.g. the automatic registration of Resource Adapters through Eclipse extension points - have to be implemented manually.

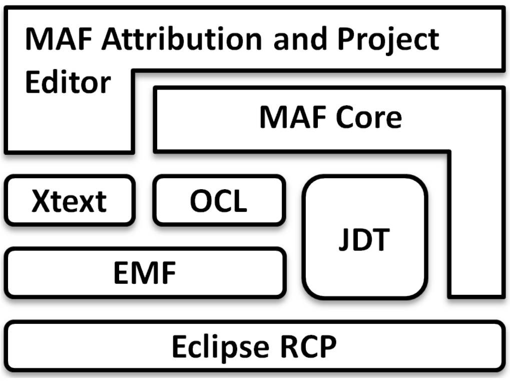


Figure 7.12.: The MAF component stack.

as a representational format for managing their internal data structures. Separated from the EMF stack are the Java Development Tools (JDT) compilers and editors which comprise Eclipse's IDE for Java development. Connection points between the modeling and the Java facilities exist, for example, in the usage of model-based code generation and in the invocation of Java methods from OCL/QVT black box operations.

The MAF Core component that implements the analysis core functionality relies on EMF to read and process analyses defined in the attribution language and to load the target (meta) models. Additionally, JDT and OCL/QVT are required to compile and execute data-flow rules specified in one of these languages. The component at the top represents MAF's Integrated Development Environment which in turn employs Xtext to implement editing capabilities for analysis specification and configuration and uses MAF Core to enable the debugging and statistical evaluation of defined analyses.

8. Implementing Flow-based Model Analysis

This chapter describes a realization of the theoretical concepts explored in Section 7.3. For this purpose, we will develop facilities for the specification, management and execution of analyses along the design goals listed in Section 7.2. The reference implementation, the Model Analysis Framework (MAF), is available from:

http://code.google.com/a/eclipselabs.org/p/model-analysis-framework/

Section 8.1 focuses on the central module of the framework, the MAF Core evaluator. This component implements the functionality of the Analysis Core (cf. Section 7.3.1) and thus provides the ability to instantiate and solve data-flow analysis specifications for models. MAF Core supports the loading of analysis resources into internally managed repositories, offers advanced logging capabilities for collecting and evaluating statistical information and exposes interfaces for the integration of custom rule specification languages and different solving strategies.

Assistance for the development of analysis specifications is provided through dedicated tooling which is the subject of Section 8.2. Text-based creation and editing of specifications is supported through a DSL editor generated from an Xtext mapping between the attribution metamodel (cf. Section 6.1) and the concrete syntax (cf. Section 6.2). This functionality is combined with embedded Java and OCL editors to enable the context-sensitive editing of data-flow rules.

For convenience reasons, all information which is required for the execution of an analysis can be encoded in a project configuration (cf. Section 7.3.2). This encompasses the input artifacts as well as the directives that govern the process of executing an analysis by imposing restrictions on the set of attributes (cf. Section 6.5). This concept, along with the corresponding tooling, the MAF project editor and the Project Set metamodel, is presented in Section 8.3.

The basic principles of the Model Analysis Framework, i.e. the facilities for analysis specification and project configuration, have been published in [SB11].

8.1. MAF Core

MAF Core is the central component of the MAF framework. It implements a highly versatile and configurable DFA solver, complemented by facilities for resource management, debugging and statistical evaluation. Its extensible design, which reflects the principles described in Section 7.3, facilitates the customization of the framework for different technological domains. This enables the integration of analysis capabilities into (existing) third-party applications. In this section, we will detail the overall architecture and the inner workings of the different functional modules.

In Section 8.1.1, we outline the architectural design of MAF Core, the framework's realization of the Analysis Core component. The functionality of the Resource Repository is described in Section 8.1.2. Because of the importance of the Instantiation concept in the context of the evaluation process, the management and the properties of this resource type are explained in more detail in Section 8.1.3. Finally, the invocation of data-flow rules and the configuration and execution of the evaluation process are the subjects of Sections 8.1.4 and 8.1.5 respectively.

8.1.1. Architecture

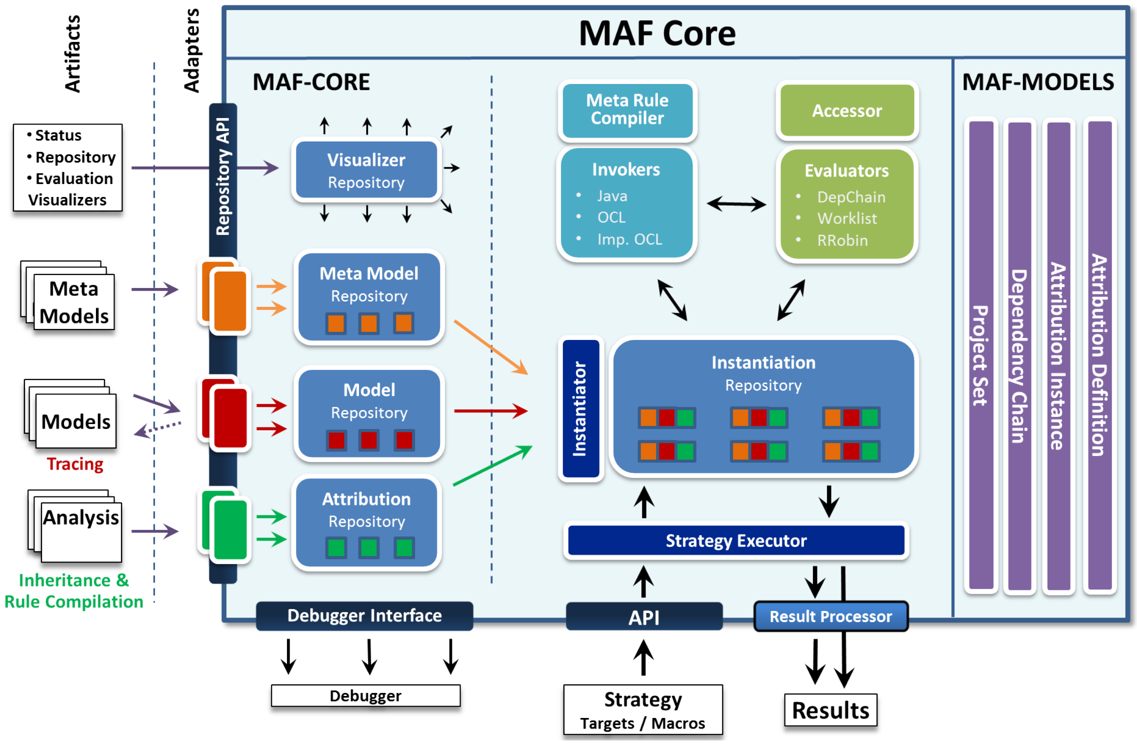


Figure 8.1.: Architecture of the MAF-Core component.

Implementation-wise, the MAF Core component has been split into two (Eclipse) projects, MAF-Core and MAF-Models. While the former implements management facilities as well as the actual DFA solver, the second project serves as a container for Ecore metamodels, i.e. the abstract syntax of the associated domain-specific languages, and the corresponding model and edit code generated by EMF (cf. Section 7.1.2).

Overall, four metamodels have to be implemented. Two of those correspond to the language definitions for attribution specification (AttrMM) and instantiation (AttrM) as described in Sections 6.1.2 and 6.3.1. An additional metamodel is required for the internal representation of the dependency chain data structure (cf. Section 6.5.4).

Finally, the structural composition of Project Set definitions is also based on a modeling language (cf. Section 8.3.1). Aggregating the corresponding Ecore files and model code in a single plugin simplifies access and enables reuse - both internally by other MAF components and by external applications (e.g. custom IDEs).

Based on the defined modeling languages, the MAF-Core (Eclipse) project realizes the functional aspects of the concepts presented in Section 7.3.1. Its internal structure, shown in Figure 8.1, can be roughly divided into three parts: The repository system that implements the resource management facilities, the notification component (Visualizer) and the modules that enable the execution of flow-analyses.

Parameter	**Description**
logLevel	Sets the logging level for the framework
threadCount	Determines how many threads are allocated for work queues (only affects parallelized algorithms)
synchronizeResources	Determines whether resources should be synchronized automatically (e.g. if a metamodel changes, all depending models, attributions and instantiations are reloaded as well)
parametersAutoreset	If set to true, changing parameters automatically reloads affected resources
autodisposePolicy	Determines whether adapters/loggers are automatically disposed when they are removed from the repository

Table 8.1.: Parameters that configure the initialization of a MAF Core instance.

MAF allows the creation of multiple instances of the core component, each of which possesses its own model and analysis repositories and can be configured independently from other instances. For this purpose, the instantiation process returns a handle to the respective framework instance which, in turn, provides access to all included functional modules through a well-defined API. Generally, there are two approaches for instantiating the framework:

- The class MAFCore implements static initialization methods (cf. Appendix D.1). To instantiate MAF Core, a set of parameters (called CoreParameters, cf. Table 8.1) is required. Optionally, the framework constructor can be supplied with a set of status and debug listeners which monitor the initialization process and subsequent actions of the framework.

- If Project Set configurations are used, the framework has to be instantiated using the methods provided by the MAFProjectSetInterface class (cf. Appendix D.8). In this case, the required parameters are read from the supplied Project Set model.

Because the framework core is designed to run as a background service, its modules and functions possess different lifecycles, the most important of which are outlined in Figure 8.2. In this example, MAF Core is instantiated (either manually or by using a Project Set configuration). Then, the required artifacts are loaded into the model repositories using resource adapters. By applying the instantiation semantics

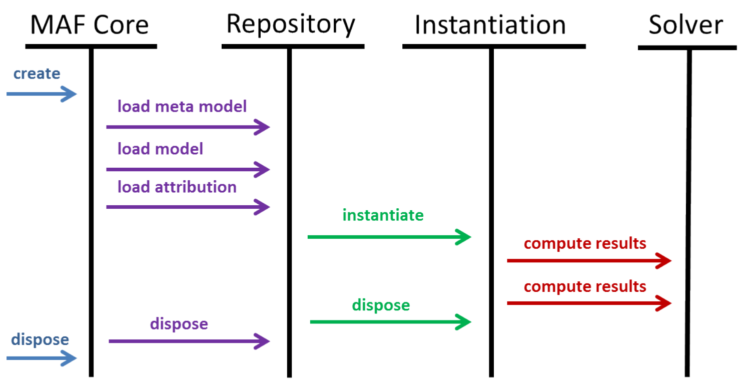


Figure 8.2.: The lifecycles of the framework's components.

described in Section 6.3, a new Instantiation resource can be created from a combination of a metamodel, a model and an attribution. This object can be passed to the solver along with a set of evaluation requests and subsequently acts as a storage container holding the hitherto computed results. These steps can be carried out repeatedly, reusing the same MAF Core instance. The resources, as well as the whole framework, can be disposed once they are no longer needed.

A comprehensive list of the most important API interfaces of the Model Analysis Framework is provided in Appendix D.

8.1.2. Resource Management

The types of artifacts which are primarily involved in the preparation and the execution of an analysis are shown as rectangles on the left hand side of Figure 8.1. The relevant types consist of the target models, their corresponding metamodel definitions and the attributions. Usually, the internal structure of an artifact depends on the technical domain for which the analysis should be implemented. This applies both to the employed MDE framework (e.g. MOF vs. KM3) and to the way this information is stored, i.e. the respective file format. To circumvent the problem of diverse input formats, we introduced the concept of Resource Adapters which act as an interface between the input artifacts and their representation as resources inside MAF's repository system.

Overall, there exist four types of adapters which are managed by the Resource Repository. Three of those correspond to the three types of input artifacts while the forth represents the Instantiation concept. The general structure of Resource Adapters is very similar, albeit with certain adaptions for the respective input type. For example, attribution adapters possess the ability to statically extend attributions so that they reflect the inheritance relationships in the corresponding metamodel (cf. Section 6.3.3). Additionally, they can make use of registered rule invokers to precompile the data-flow rules (cf. Section 6.4.1 and Section 8.1.4). Model adapters, on the other hand, have to establish and store a mapping between the elements from the model's source artifact and its target representation inside MAF. In some cases, it might be necessary for a single adapter to load multiple artifacts at once

Parameter	Description
staticRuleCreation	Determines whether rules are parsed/compiled during the loading of an attribution or on demand during the evaluation
inheritancePolicy	Determines the inheritance policy used for the attribution. Inheritance can be disabled, set to follow a strict top-down pattern and set to allow redefinition at subclasses
javaRuleClasses	The classes for external java rules in the format <javaruleclassid, classpath>
mockJavaRules	Automatically generates empty implementations of Java rules for testing purposes

Table 8.2.: Parameters required for loading an attribution.

and merge them into a single, internal resource. This is, for example, required if a model is based on multiple metamodels or if analyses are built incrementally, reusing attributes defined in other attributions.

The instantiation of a resource adapter requires the following set of parameters:

- Most importantly, the adapter must be provided with references to the source artifacts that should be loaded. Because a single adapter may have to merge more than one artifact and each artifact can be defined in a different format, this information is given in the form of a map. This data structure uses the concrete type of the artifact (e.g. the file format) as key. The entry's value encodes a type-dependent reference to the source element, e.g. a file path or an EMF model object which is already present in memory.

- Secondly, it is necessary to specify an id that uniquely identifies this adapter inside the framework instance's Resource Repository.

- If the artifact depends on other resources, their repository ids have to be specified as well. For example, for model and attribution adapters, it is necessary to provide the id of the metamodel adapter that contains their corresponding metamodel resource. An Instantiation requires references to the metamodel, model and attribution adapters from which it should be created.

- The attribution adapter additionally requires a separate set of parameters called AttributionParameters. These parameters control, amongst other settings, the inheritance policy and the precompilation of data-flow rules. They are listed in Table 8.2.

- Instantiation-specific parameters are described in Section 8.1.3.

The adapters themselves do not implement the functionality to import artifacts of different source types. Instead, the actual loading process is deferred to other components, so-called Resource Loaders. While only one adapter exists for each resource type, multiple Resource Loader implementations are used to handle different input formats. For this purpose, a loader implements a method loadIntoEMFResource which is invoked by the corresponding adapter with the reference to the source artifact. This method is responsible for processing the artifact and converting it into a suitable internal representation. The loaded resource is then handed back to the calling adapter which combines all loaded artifacts in a single resource. In the case of model adapters, the Resource Loader also has to build a map which connects the elements of the internal representation to their original form in the source artifact so that analysis results can be connected to their respective input elements. By default, MAF provides loaders for importing (meta) models from files serialized as XMI/Ecore and from EMF resources[1] which are already present in memory. The adapter interfaces are documented in Appendix D.2.

The loading process is started when an adapter is put into the Resource Repository and is repeated every time the contents of the adapter have to be updated. This happens if a connected resource, e.g. the metamodel of a model, is reloaded (depending on the state of the synchronizeResources core parameter) but the update may also be invoked manually. To load the artifacts, the respective adapter iterates over all provided inputs, determining their artifact type and invoking the corresponding Resource Loader for this type. From the loaded EMF resources, a merged representation containing all converted artifacts is incrementally built. Providing support for new input formats therefore only requires the implementation of custom Resource Loaders. These have to be registered with the framework, e.g. using Eclipse's extension point mechanism.

The Resource Repository can be accessed externally via the API of the respective framework instance. This interface exposes the methods necessary for triggering the loading of artifacts and to store adapters in the repository (cf. Appendix D.4). Additional functions enable a lookup of resources based on their ids and the removal of adapters from the repository. Based on the value of the synchronizeResources core parameter, dependent resources are automatically refreshed or removed after these actions. All activities of the repository are also reported to loggers via the repository listener interface (cf. Appendix D.5).

8.1.3. Managing Attribute Instantiations

Technically, Instantiations can be classified as a forth adapter type. Like the other adapters, they are also managed by the repository system. In contrast to the other types, the creation of an Instantiation does however not require the specification of source artifacts as it solely relies on resources which have already been loaded into the repository. Their creation can be triggered through corresponding API calls (cf. Appendix D.4). The required input for this process consists of the ids of a metamodel, an attribution and a model adapter along with a set of InstantiationParameters (cf. Table 8.3).

During the initialization process, the Instantiation prepares the rule invokers, the selected solver and an EMF resource that serves as a container for the attribute instances. Depending on the setting of the instantiationPolicy parameter, the attributes

[1]EMF models which are loaded from memory into the MAF repository are copied first to avoid problems that could arise if either MAF or another application modifies the shared resource.

Parameter	**Description**
instantiationPolicy	Determines whether attribute instances are created statically or on demand and whether they are immediately initialized
autoDeletePolicy	Determines whether an instantiation is removed automatically after it has been evaluated.
synchronizeEvaluation	Determines whether the evaluation of all attributes should be triggered every time the instantiation is refreshed
synchronizeAttributions	Limits the evaluation synchronization to attributions with the specified ids
problemMarkerPolicy	Determines whether Eclipse problem markers are added/replaced for violated constraints
evaluationVisualizers	The Result Processors that are called to examine the results of the evaluation
validateReferences	Detect invalid call-by-reference modifications of attribute values
blockStable	Determines whether stable results are blocked for reevaluation (i.e. treated as constant values)
maxRuleInvoke	Abort evaluation after given amount of rule executions
evaluatorType	The DFA solving algorithm used for the evaluation process
evaluatorParameters	Parameter set specific to the chosen solving algorithm
measurePerformance	Record detailed statistical information during fixed-point evaluation
evaluatorDebugMode	Output debug information about the evaluation process

Table 8.3.: Parameters required for instantiating an attribution.

are either immediately instantiated and assigned their respective initialization values or the instantiation is deferred to a later stage (*dynamic instantiation*).

Through the autoDeletePolicy parameter, the repository can be directed to dispose an Instantiation (and optionally the resources from which is has been created) once the analysis process has been completed. synchronizeEvaluation triggers the automatic evaluation of the contained attributes on creation. If this parameter is set, the analysis is carried out immediately after the initialization process and also if one of the Instantiation's input resources is reloaded or if one of its parameters changes. With synchronizeAttributions, the (re)evaluation can be restricted to a subset of attributions.

The Instantiation adapter also implements functionality which is required for accessing and manipulating attribute instances but is independent of a specific solving algorithm. It provides interfaces for manual attribute instantiation and for invoking their associated initialization and iteration rules. All requests and callbacks from the Strategy Executor, the Attribute Accessor and the selected Data-flow Solver are relayed through and coordinated by this component. Instantiations therefore provide the generic functionality required for the preparation and execution of a data-flow analysis. By activating the blockStable setting, stable instance results computed by the execution of a previous Evaluation Target are treated as constant inputs for subsequent evaluations, thus avoiding the recomputation of already available results. Enabling validateReferences provides a check for invalid *call-by-reference modifica-*

tions of attribute results by data-flow rules[2]. Furthermore, an Instantiation records statistical information about the frequency and the execution time of different actions during the evaluation process. The measurePerformance parameter enables the collection of more detailed information but may increase the time required for carrying out computations.

8.1.4. Data-flow Rule Invocation

A Rule Invoker for a specific rule implementation language consists of two parts: A preprocessor, used by attribution adapters for the precompilation/parsing of the rules, and the actual invoker. During the loading process, the attribution adapter traverses all defined data-flow rules and passes each rule to the preprocessor of the rule's respective target language using the preprocessor's addRule() method. Only after all rules have been assigned to their corresponding processors, the actual precompilation is started by executing finishRulePreparation(). This approach improves the performance since it reduces the overhead by triggering the compilation step only once for each processor type. The actual invokers are created at a later stage, namely during the initialization of an Instantiation. They are supplied with the precompiled rules stored inside the Instantiation's associated attribution adapter.

By default, MAF contains four types of invokers, respectively implementing the preprocessor/invoker design in the following fashion:

Java (source code)
If data-flow rules are specified in the form of Java source code, this code has to be compiled into a class file. For this purpose, the preprocessor constructs a Java class containing all rules as Java methods. Each rule is provided with a fixed method signature that takes the accessor object, the respective attribute definition and the context object as its parameters. Once all rules have been added, the class containing the aggregated rules is compiled using Eclipse's in-memory compiler. References to the compiled methods are then extracted from the resulting class using Java's reflection API. Invoking a rule then only requires to locate the corresponding method handle and executing it, again making use of Java's reflection capabilities. The Instantiation is responsible for supplying the correct method parameters for these calls.

Java (class files)
The process for invoking methods from class files is essentially the same as for methods compiled from source code. The difference lies in the preparation step which locates and loads the class file using MAF's advanced class loader capabilities which are able to locate classes in standalone Java applications as well as in Eclipse plugins.

[2]By definition, a data-flow rule is not allowed to modify values of other attribute instances. This problem occurs if a rule requests a result in the form of a non-primitive Java object (i.e. a reference) from another attribute and changes it.

OCL

The parsing of OCL rules requires an extension of the OCL evaluation environment. The environment provides the context (e.g. variable bindings and available operations) in which the constraints are parsed and interpreted. In order to be able to request attribute instance values located at model elements, each metamodel class has to provide additional operations for accessing attributes that have been annotated at this class (cf. Section 6.4.2). For this purpose, the preprocessor analyzes which attributes extend which classes. For each attribute available at a class, an accessor operation with the name of the corresponding attribute is injected into the evaluation environment for this class type. If, for example, an attribute *attr* has been defined for the metamodel class *class*, attribute instances at model elements can be accessed via $element_{instanceof(class)}.attr()$. The Java implementation of these operations automatically redirects these calls to the Attribute Accessor object which generates corresponding requests to the Data-flow Solver module.

QVT

Conceptually, the process involved in the preparation of QVT rules is a combination of the Java source code compilation and the method used for preparing OCL rules. During the preprocessing step, a single QVT library is built that contains all rules as query definitions as well as custom operations which extend the metamodel's classes with accessor methods. This library is then parsed using the QVT framework's library compiler. Instead of extending the evaluation environment, support for attribute accesses is provided through black box functions, thereby making use of QVT's designated extension mechanism.

In summary, providing support for custom rule specification languages requires the implementation of a Rule Invoker which implements the functionality for both the precompilation and the rule invocation steps. Precompilation is controlled by attribution adapters while the execution of rules is triggered by an Instantiation. Rule implementations also have to be supplied with a language-specific interface which enables them to request input values by invoking the Attribute Accessor.

8.1.5. The Data-flow Solver

The data-flow solvers for fixed-point computation realize the core functionality of the Model Analysis Framework by implementing the algorithms required to compute results for dynamic DFA equation systems (cf. Section 6.5). The architectural design of the solver infrastructure of MAF provides a highly configurable and flexible framework for the execution and the statistical evaluation of different algorithms. For this purpose, the architecture sources out all functionality not essential to the actual computation process to other modules. As outlined in Figure 8.1, computing results requires the interaction of three components: The Rule Invokers, a Data-flow Solver and the target Instantiation. Furthermore, the overall evaluation process is directed by the Strategy Executor.

To understand this methodology, we first have to examine how a flow analysis can be configured and executed. Details about the solving process are encoded in an Evaluation Strategy which consists of a list of Evaluation Directives (cf. Section 7.3). The Evaluation Strategy is passed as input to the Strategy Executor API interface of a MAF Core instance. Optionally, a list of Result Processors which have already been registered with the current framework instance can be supplied. These processors will be called automatically on completion of the analysis to handle the results. The strategy and the list of relevant Result Processors can either be provided via the API or read from a Project Set definition.

The Strategy Executor processes the list of directives one by one, executing the contained Evaluation Target and Evaluation Macro directives. From the partial results returned by each executed target (SimpleEvaluationResult), an AggregatedEvaluationResult object is built representing the unification of the results of the overall evaluation process (cf. Appendix D.5). This data structure consists of several maps that contain the computed values for attribute assignments and attribute constraints, the latter separated into sets of passed and violated constraints. Furthermore, it contains the trace information generated by the model adapter that is required to relate the results to their corresponding elements of the original artifact. Finally, it also provides access to statistical information about the evaluation process such as the total amount of executed rules and the time spent on different parts of the analysis process. After the strategy execution has finished, the AggregatedEvaluationResult object is passed to the specified Result Processors.

Now, we will take a closer look at the two types of Evaluation Directives. The request to execute a DFA on (a subset of) the attributes contained in an Instantiation is encoded in an Evaluation Target. The parameters of a target can be grouped as follows:

Instantiation Target
The first set of parameters specifies the attributed model (i.e. Instantiation) for which the target should be executed. For this purpose, three repository ids referencing a metamodel, a model and an attribution Resource Adapter have to be provided. This information uniquely identifies the corresponding Instantiation in the repository if it already exists or enables the creation of a new Instantiation adapter from the selected resources otherwise.

Attribute Instance Selection
The remaining parameters configure the demand-driven computation by narrowing down the set of attribute instances of the selected Instantiation that should be computed by the execution of this particular target. There are multiple ways of achieving this effect:

- If no further limitations are given, all attribute instances defined in the Instantiation's attribution(s) will be evaluated.
- Since the merged AttributionCollection resource of the corresponding adapter may be composed of multiple attribution models, it can be convenient to limit the computation to a predefined list of attributions. With this

feature, one can focus on the execution of a specific analysis scenario (provided that the attributions are structured in a suitable way). For example, the case study described in Section 10.1 contains multiple use cases which have been split into separate attributions. By specifying a list of relevant attributions, all scenarios can be loaded into a single attribution adapter. If the user then chooses to only execute a single use case, it is possible to limit the computation process to the corresponding attribution(s).

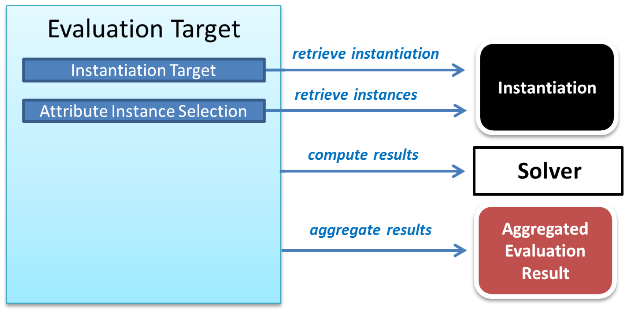
- On a more fine-grained level, it is also possible to compute instances for specified attribute ids, thereby limiting the calculation to a subset of attributes defined inside a single attribution. Additionally, this can also be used to combine the analysis of multiple attributes spanning several different attributions.
- Another way to address a specific set of instances relies on the type of the attributed class. This means that all instances of attributes that have been annotated at the given class(es) will be computed. This method of selection is useful if one is interested in the evaluation of all elements corresponding to a certain metamodel type.
- Finally, it is also possible to provide a set of model elements for which attribute values should be calculated. This is important if the analysis should focus on the evaluation of a particular model element. The typical usage scenario would be the selection/modification of a specific object in the user interface of the target application which then must be reevaluated.

At this point, it is important to note that an Attribute Instance Selection only provides a starting set for the DFA solver. Due to the *dependency discovery* feature, this set may be extended during the solving process if attribute instances in the initial selection depend on values of other attributes. This approach therefore simplifies the execution of analyses as users only need to supply the instances whose values are currently of interest and the respective solving algorithm automatically extends this selection to the smallest set of relevant elements required to compute the desired results.

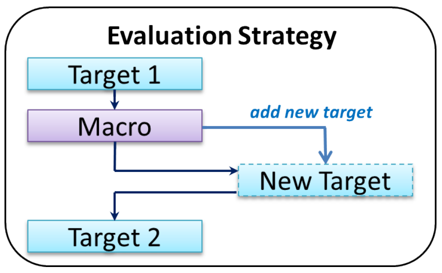
The steps involved in the execution of a target are shown in Figure 8.3(a). They consist of:

1. **Acquiring the Instantiation Target**
   In the first step, the Instantiation that corresponds to the Instantiation Target parameters listed in the Evaluation Target is retrieved from the repository (or created if it does not yet exist).

2. **Collecting the Attribute Instances**
   Then, the list of relevant attribute instances is built depending on the target's Attribute Instance Selection. In this step, the instances are created while the



(a) Executing a target



(b) Dynamically changing the strategy using macros

Figure 8.3.: Execution of Evaluation Directives through the Strategy Executor interface.

initialization follows the chosen instantiationPolicy (*dynamic instantiation/initialization* of additionally required attributes is handled by the respective fixed-point algorithm).

3. **Computing the Results**
   The selected Data-flow Solver (evaluatorType) is invoked with the set of selected instances. It is responsible for executing their initialization and iteration rules in a valid order and for managing dynamically discovered dependencies to additional attributes (cf. Section 6.5).

4. **Result Processing**
   Once the computation has finished, a SimpleEvaluationResult object is constructed that contains the results and statistical information recorded by the Instantiation and the solver. The StrategyExecutor combines results from multiple targets into a single AggregatedEvaluationResult. Additionally, depending on the problemMarkerPolicy setting, Eclipse problem markers are created for violated constraints.

The second Evaluation Directive type is the Evaluation Macro. A macro consists of a method which is called by the Strategy Executor once the processing of the list of directives reaches the position of this element. Apart from providing the possibility to interweave the analysis process with custom functions, macros also may alter the structure of the remaining part of the executed strategy. For this purpose, the macro method is invoked with the hitherto computed results in the form of an AggregatedEvaluationResult and the list of the succeeding directives in the strategy. The macro can modify this list by adding or removing directives which influences the subsequent part of the evaluation process as shown in Figure 8.3(b). The use case described in Section 10.1.4 makes use of this functionality to recursively apply the analysis to nested cycles: If a cycle has been identified, the model has to be modified so that the substructure can be analyzed. In this case, new directives are dynamically created and added to the Evaluation Strategy to adapt the model and recursively repeat the analysis on the therein contained substructures.

We will now shortly describe the properties of MAF's solver infrastructure. Each implementation of a fixed-point algorithm has to implement two functions (cf. Section 6.5.1). The interface between the Strategy Executor and the solvers consists of the method evaluateAttributes() which corresponds to the Analysis Entry Point. It is invoked in the step Computing the Results with the respective instance selection. The Callback Handler is realized by the method evaluateRecursiveCall() which is called by the Attribute Accessor if a data-flow rule requests an input. The requested instance is located by the accessor and passed as a parameter to the solver. The callback function of the solver is then responsible for handling the dependency between the calling and the called instance and may invoke the called instance's initialization or iteration rule (*recursive lookup*).

Parameter	**Description**
Round Robin	
eval_roundrobin_reclookup	Perform recursive lookup for attribute requests
Worklist	
eval_worklist_reclookup	Perform recursive lookup for attribute requests
eval_worklist_changesets	Processor for changesets: processor1, processor2
Dependency Chain	
eval_depchain_phase1_parallelize	Parallelize phase 1 of the evaluation process
eval_depchain_phase1	Chain builder for phase 1: recursive, iterative1, iterative2
eval_depchain_phase2_parallelize	Parallelize phase 2 of the evaluation process
eval_depchain_phase2	Traversal strategy for phase 2: bottom-up recursive, bottom-up iterative, worklist
eval_depchain_bu_eliminate	Bottom-up processor eliminate (only for bottom-up traversal)
eval_depchain_bu_postpone	Bottom-up processor postpone (only for bottom-up traversal)
eval_depchain_bu_start	Starting point for bottom: complete, leaves, changed (only for bottom-up traversal)
eval_depchain_wl_adddiscovered	Worklist processor add discovered (only for worklist traversal)

Table 8.4.: Solver-specific parameters.

The different solving strategies can be configured via the evaluatorParameters which are part of the InstantiationParameters. The available parameters are listed in Table 8.4, sorted by algorithm. The round-robin and worklist algorithms on the one hand and the dependency chain algorithms on the other hand each share a set of common parameters:

- Using the parameters eval_roundrobin_reclookup/eval_worklist_reclookup, the detection of dependencies to attributes outside the set of initially selected instances can be improved. In recursive lookup mode, a callback immediately invokes the rule for the requested instance to identify multiple unknown dependencies during a single callback.

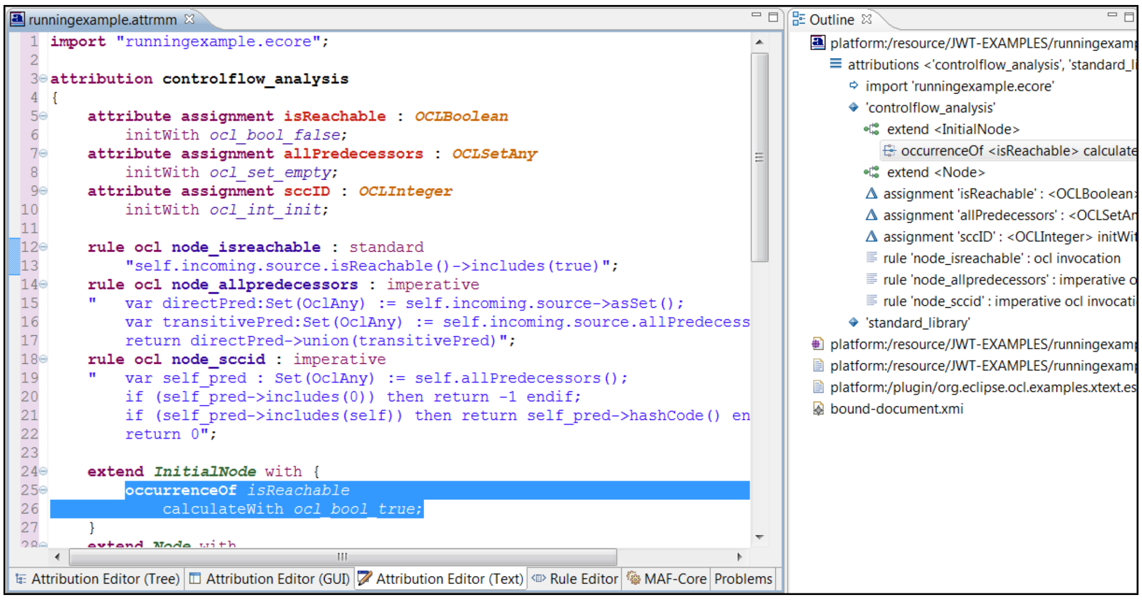
- The parameters eval_worklist_changesets, eval_depchain_phase1 and eval_depchain_phase2 can be used to switch between different strategies for separate phases of these algorithms.

8.2. Analysis Editor

Analysis specification in the Model Analysis Framework IDE is supported through a dedicated Analysis Editor. This component integrates with the Eclipse workbench and is automatically activated when an attrmm file is selected for editing. The Analysis Editor features text-based editing capabilities for attributions. It has been implemented using the Xtext parser/editor generator and is based on a mapping between the attribution metamodel AttrMM and a textual domain-specific language. The editor also supports in-place editing of OCL, QVT and Java code to simplify the definition of data-flow rules. The validation of the specifications is carried out using a MAF Core instance that runs in the background and provides - amongst other services - the facilities required for the compilation/parsing of analysis definitions.

The capabilities of MAF's Analysis Editor are described in the following sections: The different methods for the definition of attributions are illustrated in Section 8.2.1 while the aspects of rule specification and management are the subject of Section 8.2.2.

8.2.1. Attribution Editors


runningexample.attrmm

```
import "runningexample.ecore";

attribution controlflow_analysis
{
    attribute assignment isReachable : OCLBoolean
        initWith ocl_bool_false;
    attribute assignment allPredecessors : OCLSetAny
        initWith ocl_set_empty;
    attribute assignment sccID : OCLInteger
        initWith ocl_int_init;

    rule ocl node_isreachable : standard
        "self.incoming.source.isReachable()->includes(true)";
    rule ocl node_allpredecessors : imperative
    "   var directPred:Set(OclAny) := self.incoming.source->asSet();
        var transitivePred:Set(OclAny) := self.incoming.source.allPredecess
        return directPred->union(transitivePred)";
    rule ocl node_sccid : imperative
    "   var self_pred : Set(OclAny) := self.allPredecessors();
        if (self_pred->includes(0)) then return -1 endif;
        if (self_pred->includes(self)) then return self_pred->hashCode() en
        return 0";

    extend InitialNode with {
        occurrenceOf isReachable
            calculateWith ocl_bool_true;
    }
    extend Node with
```

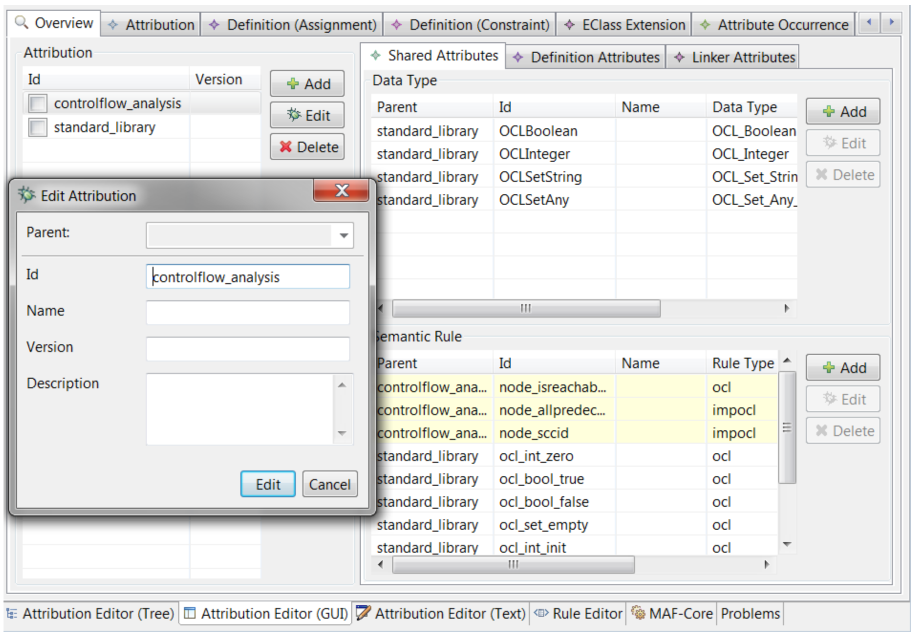
Outline

```
platform:/resource/JWT-EXAMPLES/runningexam
  attributions <'controlflow_analysis', 'standard_l
    import 'runningexample.ecore'
    'controlflow_analysis'
      extend <InitialNode>
        occurrenceOf <isReachable> calculate
      extend <Node>
      assignment 'isReachable' : <OCLBoolean
      assignment 'allPredecessors' : <OCLSetAn
      assignment 'sccID' : <OCLInteger> initWi
      rule 'node_isreachable' : ocl invocation
      rule 'node_allpredecessors' : imperative o
      rule 'node_sccid' : imperative ocl invocati
    'standard_library'
platform:/resource/JWT-EXAMPLES/runningexam
platform:/resource/JWT-EXAMPLES/runningexam
platform:/plugin/org.eclipse.ocl.examples.xtext.es
bound-document.xmi
```

Figure 8.4.: Analysis specification in the textual editor.

Figure 8.4 shows the Analysis Editor displaying the contents of an attribution model in the textual editor component. The attribution controlflow_analysis shown in the screenshot corresponds to the running example from Section 6.1. The Xtext editor has been automatically generated from the grammar presented in Section 6.2 which connects the metamodel elements to the textual syntax. It supports features commonly expected from modern DSL editors such syntax highlighting, code completion and autoformatting. Since attributions extend existing metamodels, the implementation generated by Xtext had to be extended with the ability to load target metamodels from Ecore files through an import statement. This way, cross-references to classes from imported metamodels can be directly specified inside the textual description of the respective analysis.

Further integration with the Eclipse IDE is provided through the utilization of Eclipse's outline view as seen on the right hand side of the screenshot. This window displays the overall structure of the currently selected attribution model, thereby giving the developer a quick overview over the defined elements. The textual representation of the model and its outline view are synchronized so that each change to one of these formats is immediately reflected in the other. Furthermore, selection is also synchronized between the outline and the textual editor so that after selecting an element, the corresponding element in the other view is automatically highlighted as well. This allows, for example, to quickly navigate to a target element in the editor by selecting it in the outline view.



Attribution

Id	Version
controlflow_analysis	
standard_library	

Edit Attribution

Parent:	
Id	controlflow_analysis
Name	
Version	
Description	

Edit Cancel

Data Type

Parent	Id	Name	Data Type
standard_library	OCLBoolean		OCL_Boolean
standard_library	OCLInteger		OCL_Integer
standard_library	OCLSetString		OCL_Set_Strin
standard_library	OCLSetAny		OCL_Set_Any_

Semantic Rule

Parent	Id	Name	Rule Type
controlflow_ana...	node_isreachab...		ocl
controlflow_ana...	node_allpredec...		impocl
controlflow_ana...	node_sccid		impocl
standard_library	ocl_int_zero		ocl
standard_library	ocl_bool_true		ocl
standard_library	ocl_bool_false		ocl
standard_library	ocl_set_empty		ocl
standard_library	ocl_int_init		ocl

Figure 8.5.: Analysis specification in the form-based editor.

While the textual DSL presents a convenient way for creating analysis specifications, in some cases other types of editors are more suitable for this task. For this purpose, the Analysis Editor includes a separate form-based interface which is shown in Figure 8.5. This component can be accessed by selecting the GUI tab at the bottom of the IDE's main window. It provides several views on the loaded attribution model that focus on different constituents of the attribution such as occurrences, rules etc. The creation, modification and removal of elements is supported through wizards which enable an easy specification of the required information and validate the user's input before the changes are applied to the model.

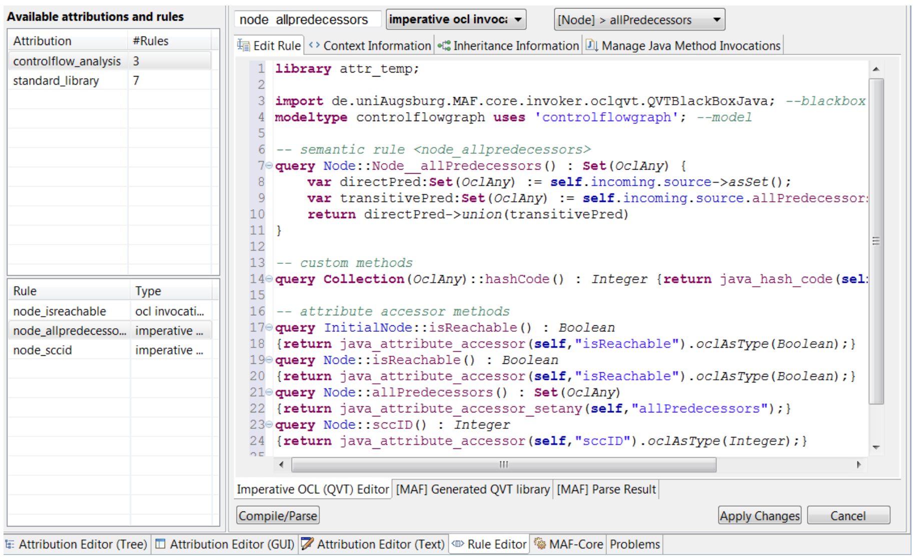
Finally, as a third option for modifying attributions, a tree-based EMF-style editor is included (accessible via the left-most tab at the bottom of the window). In order to synchronize the contents of the loaded specification between the different representations, all three editors as well as the outline view use a single internal EMF representation of the underlying model. Through listener interfaces, changes are immediately propagated to all relevant components which update their representations accordingly.

8.2.2. Rule Editor

The Analysis Editor combines the Attribution Editors with support for the context-aware creation and manipulation of data-flow rules through the Rule Editor component. This is necessary because rules are typically given as strings which are compiled/parsed once the specification is loaded by MAF Core's repository system. While it would be possible to define the rules using the corresponding language tools such as Eclipse's Java or OCL editors, this approach has several disadvantages: Firstly, it would require to continuously switch between the language editors and the Attribution Editor, repeatedly copying the data-flow rules in the language editor and inserting them in the textual or graphical representation of the attribution model. It is obvious that this does not represent a satisfying solution for a continuous, intuitive workflow for the analysis specification process. Secondly, many data-flow rules depend on the context in which they have been defined. This is due to the fact that rules are interpreted in the context of the metamodel class(es) at which the rules' occurrences are annotated. For example, OCL rules may access class attributes and references through the `self` object reference which is only available in the variable environment of the respective metamodel class. If rules are edited using a separate component, it would be necessary to configure this property manually. Finally, an integrated rule editor can identify problems caused by inconsistent definitions during the specification process rather than in a later step, e.g. when the analysis is loaded into the evaluation framework.

The Analysis Editor addresses these problems by integrating rule editing capabilities for MAF's default implementation languages. For this purpose, Java, OCL and QVT editors have been embedded into the tooling's IDE. These components can be used to quickly create and modify data-flow rules for existing attributions and are able to automatically supply the correct context to simplify the specification process. Furthermore, the Analysis Editor uses an instance of MAF Core to process the rules using MAF's internal facilities in order to immediately provide feedback about the validity of a definition.

This is demonstrated in the example shown in Figure 8.6. In this screenshot, the rule for computing the iteration value of the allPredecessors attribute at Nodes, defined in the attribution controlflow_analysis, is selected for editing. Since it is specified in imperative OCL, the embedded QVT editor is used to display its contents. To enable context-aware editing, the actual code is embedded into a library



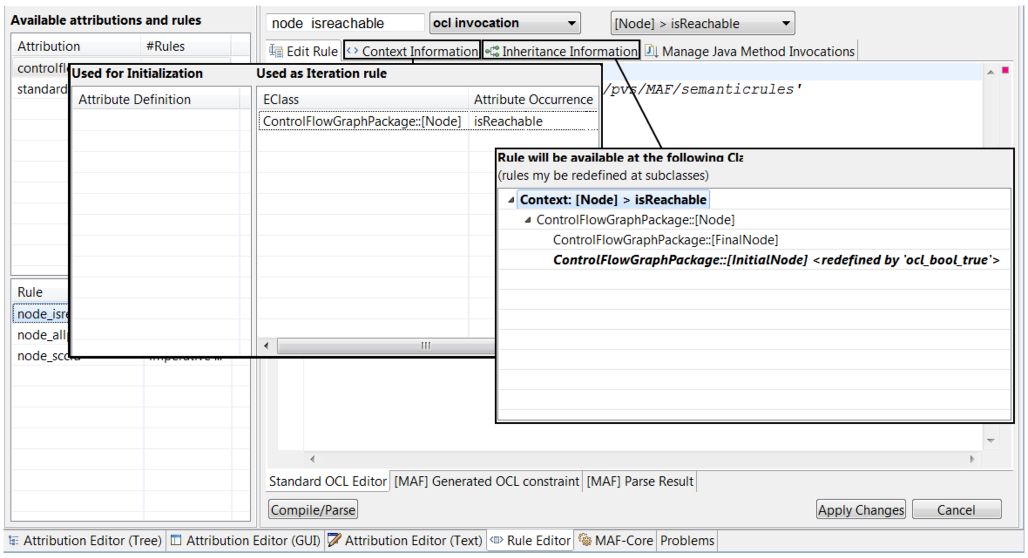
Attribution	#Rules
controlflow_analysis	3
standard_library	7

Rule	Type
node_isreachable	ocl invocati...
node_allpredecesso...	imperative ...
node_sccid	imperative ...

```
library attr_temp;

import de.uniAugsburg.MAF.core.invoker.oclqvt.QVTBlackBoxJava; --blackbox
modeltype controlflowgraph uses 'controlflowgraph'; --model

-- semantic rule <node_allpredecessors>
query Node::Node__allPredecessors() : Set(OclAny) {
    var directPred:Set(OclAny) := self.incoming.source->asSet();
    var transitivePred:Set(OclAny) := self.incoming.source.allPredecessor:
    return directPred->union(transitivePred)
}

-- custom methods
query Collection(OclAny)::hashCode() : Integer {return java_hash_code(sel:

-- attribute accessor methods
query InitialNode::isReachable() : Boolean
{return java_attribute_accessor(self,"isReachable").oclAsType(Boolean);}
query Node::isReachable() : Boolean
{return java_attribute_accessor(self,"isReachable").oclAsType(Boolean);}
query Node::allPredecessors() : Set(OclAny)
{return java_attribute_accessor_setany(self,"allPredecessors");}
query Node::sccID() : Integer
{return java_attribute_accessor(self,"sccID").oclAsType(Integer);}
```

Figure 8.6.: Editing a data-flow rule using the embedded QVT editor.

generated by the precompiler of MAF Core's QVT Rule Invoker. This library provides the correct method signature consisting of the context (the metamodel class Node) and the return type (Set(OclAny)). In order to be able to access attribute instances at model objects, Attribute Accessor helper queries are automatically generated for all defined occurrences.



Used for Initialization

Attribute Definition

Used as Iteration rule

EClass	Attribute Occurrence
ControlFlowGraphPackage::[Node]	isReachable

/pvs/MAF/semanticrules'

Rule will be available at the following Cl
(rules my be redefined at subclasses)

- **Context: [Node] > isReachable**
  - ControlFlowGraphPackage::[Node]
    - ControlFlowGraphPackage::[FinalNode]
    - ***ControlFlowGraphPackage::[InitialNode] <redefined by 'ocl_bool_true'>***

Figure 8.7.: Additional information available for the rule node_isReachable.

The rule specification process is further simplified through the generation of additional information as shown in Figure 8.7: Because each rule may be used to compute the initialization and/or iteration values of different attributes, all relevant contexts of the active rule are displayed in a separate tab named Context Information. In this case, according to the analysis specification, the selected rule node_isReachable is used for the calculation of the value of the attribute isReachable at the Node class.

A separate page Inheritance Information aggregates information about the inheritance status of the attribute and the accompanying rules depending on the generalization hierarchy defined in the metamodel[3]. In the running example, occurrences of the attribute isReachable have been assigned to the classes Node and InitialNode. The inheritance page therefore indicates the availability of this attribute at these types. Since FinalNode inherits from Node and therefore implicitly possesses the attribute isReachable as well, this class is also contained in that list. Finally, the visualization indicates that, while the attribute is available at all of these classes, the currently selected rule node_isReachable is only used to compute results for instances of Node and FinalNode since InitialNode provides a redefinition of the occurrence. Because of the redefinition of isReachable at InitialNode, the computation of iteration values at elements of this type instead depends on the rule ocl_bool_true.

8.3. Project Set Editor

With the Project Set concept, analysis resources can be assembled and packaged along with fine-grained evaluation strategies into a single project configuration.

In Section 8.3.1, we describe the internal project structure and how this concept can be used to facilitate the integration of analysis capabilities into third-party applications. The subsequent sections detail the different aspects of the project configuration. In Section 8.3.2, the resource management of the Project Set Editor is outlined. The relationships between analysis resources and Evaluation Strategies is detailed in Section 8.3.3. We then give an overview over the debugging and testing capabilities of the project editor in Section 8.3.4. Finally, in Section 8.3.5, we will outline an alternative way to define Project Sets using a textual DSL editor analogous to the Xtext Attribution Editor.

8.3.1. Analysis Configuration and Execution

The structure of a Project Set is given by a metamodel which aggregates all information that is necessary to load and execute an analysis. The stated goal here is the reduction of the effort required for configuring an analysis while at the same time providing enough flexibility to control the evaluation process. This primarily encompasses tasks such as resource loading and strategy definition.

The principle behind the utilization of Project Sets is illustrated in Figure 8.8. In addition to the Java API which has been described in Section 8.1, the solver module provides a separate Project API for loading Project Set configurations. Using the Java API, resources, processors and strategies have to be handled manually which allows for a high level of control in the configuration and the management of the evaluation process. While this approach has the advantage of enabling a detailed management of all aspects involved in the execution of analyses, it can be beneficial to abstract from the inner workings of the framework and provide a unified interface for this

[3]This information is retrieved from the attribution adapter.

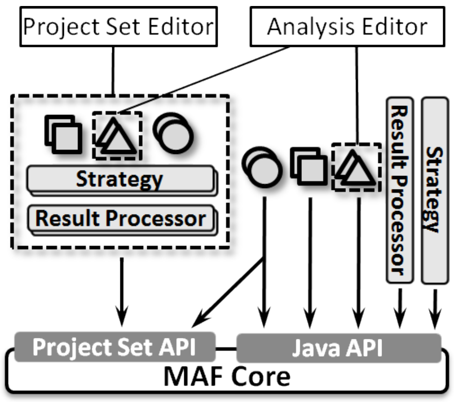


Figure 8.8.: Analysis configuration using the Java API and Project Sets.

purpose. The Project Set based configuration therefore focuses on the realization of the most common use case of loading a set of resources along with Result Processors and Evaluation Strategies and making this process easily accessible to developers.

To this end, the Project Set API exposes the required functions to initialize MAF Core based on a Project Set definition. This model references the required resources and specifies corresponding analysis configurations. Once a project has been loaded, only two additional inputs must be supplied to run an analysis: The target model which should be evaluated (symbolized by a circle in the illustration) and the id of an Evaluation Strategy. A detailed list of the interfaces of the Project Set API is given in Appendix D.8.

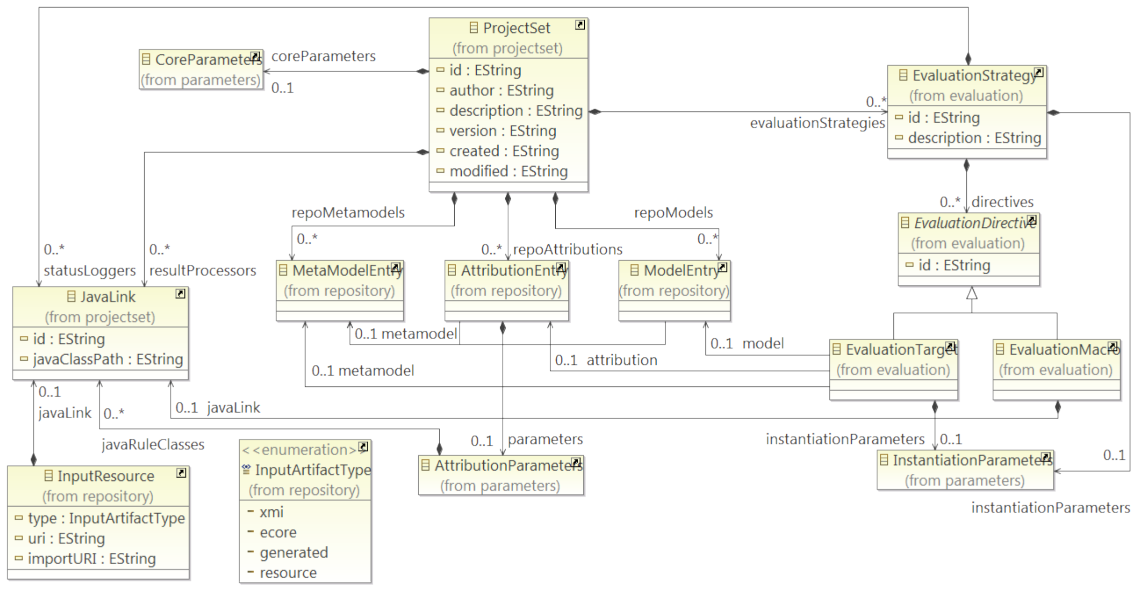

ProjectSet (from projectset)
id : EString
author : EString
description : EString
version : EString
created : EString
modified : EString
CoreParameters (from parameters)
coreParameters 0..1
EvaluationStrategy (from evaluation)
id : EString
description : EString
evaluationStrategies 0..*
0..* directives
EvaluationDirective (from evaluation)
id : EString
repoMetamodels
repoModels
0..* repoAttributions
MetaModelEntry (from repository)
AttributionEntry (from repository)
ModelEntry (from repository)
0..* statusLoggers
0..* resultProcessors
JavaLink (from projectset)
id : EString
javaClassPath : EString
0..1 metamodel
0..1 attribution
0..1 model
EvaluationTarget (from evaluation)
EvaluationMacro (from evaluation)
0..1 javaLink
0..* javaRuleClasses
0..1 parameters
instantiationParameters 0..1
InputResource (from repository)
type : InputArtifactType
uri : EString
importURI : EString
<<enumeration>> InputArtifactType (from repository)
xmi
ecore
generated
resource
AttributionParameters (from parameters)
InstantiationParameters (from parameters)
instantiationParameters

Figure 8.9.: The Project Set metamodel.

The project metamodel's essential constituents are depicted in Figure 8.9. The root element is the ProjectSet class which stores meta information about the project such as the author and the version number. Since each instance of MAF has to be configured with exactly one project definition, a ProjectSet model contains a single CoreParameters structure (cf. Table 8.1). The ProjectSet element can optionally specify global Status Loggers and Result Processors via a JavaLink referencing the implementing Java class.

Resource loading is realized through the MetaModelEntry, ModelEntry and AttributionEntry classes. From their parent class RepositoryModelEntry (not depicted in the diagram), they inherit an attribute repoID which is used to specify the repository id for storing the loaded artifacts in the repository system. Relationships between resources such as a model resource's respective metamodel entry are expressed using cross references between the related objects. Each of the three repository entry types may define links to a list of InputResources consisting of an artifact type and its location.

Evaluation Strategies, each of which consists of a list of Evaluation Directives, can be specified using the similarly named class in the project metamodel. A global InstantiationParameters object can be assigned to each strategy. This configuration is then used for all contained Evaluation Targets which do not define their own parameters. An Evaluation Macro simply stores a reference to the implementing Java class. An Evaluation Target, on the other hand, references one of each of the three resource types[4] (thereby identifying the target Instantiation) as well as a set of InstantiationParameters if the target needs to override the parameters of its parent strategy. This approach drastically simplifies the evaluation process since the creation and the lifecycle management of Instantiations are handled automatically.

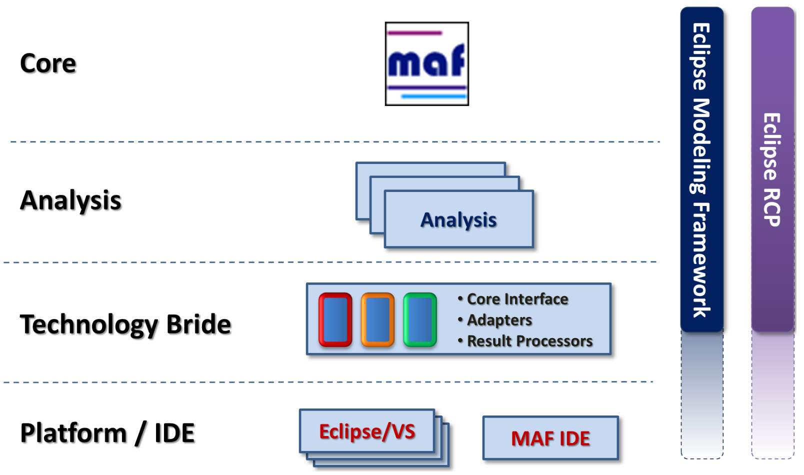


Figure 8.10.: Integrating MAF into existing tooling ecosystems.

Figure 8.10 outlines the integration of analysis capabilities into third party tools: The target and the development environments are located at the bottom of this architecture. This level therefore comprises the application which should be extended with analysis capabilities as well as MAF's Integrated Development Environment. The Technology Bride forms the "glue" between MAF and the target application and

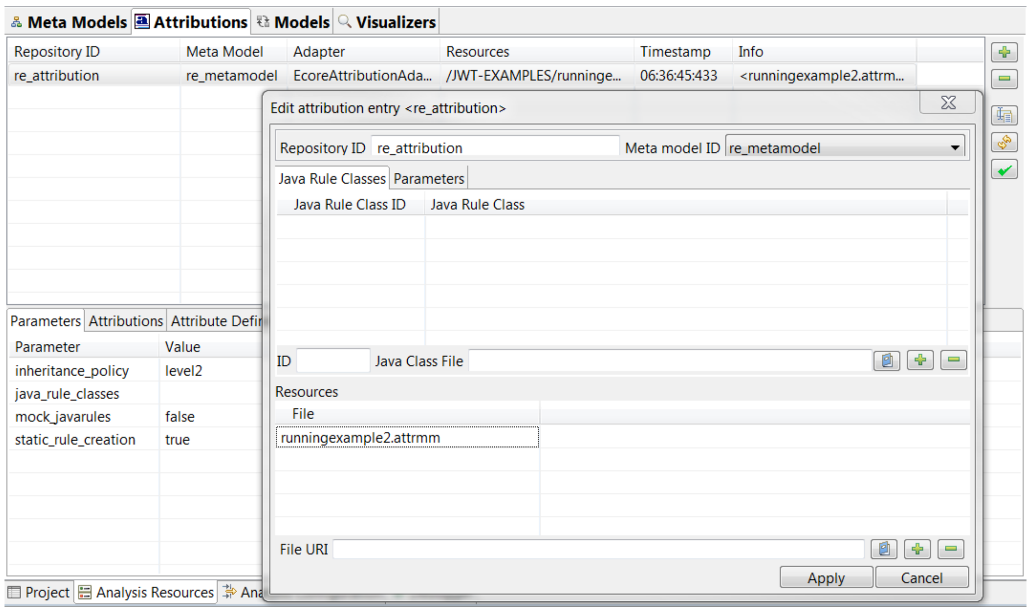
[4]Links to model resources are optional since the target model is usually provided at runtime by the external application.

consists of an implementation of the required interfaces and technology adapters. These may include, for example, model transformations and resource loaders for converting source artifacts as well as result processors which refine the raw analysis results. The Analysis layer provides the technology-independent resources such as attributions and Project Sets. Finally, the top-most layer in this diagram consists of the Model Analysis Framework which forms the basis for the execution of the analyses. On the right hand side, the relation between the MAF stack and the Eclipse RCP and EMF frameworks is depicted. The Technology Bridge forms the border between MAF-related tools and third-party environments by implementing interfaces which connect the platform to the analysis layer.

A concrete application of this concept is shown in Figure 10.20 in the context of the JWT case study which is the subject of Section 10.1.

8.3.2. Resource Management

The Project Set Editor includes a resource management system for specifying which contents should be loaded into MAF's internal repository. For each resource, the corresponding repository entry (cf. Figure 8.9) is immediately created and synchronized with the editor's internal MAF instance. This way, the validity of the specifications can be checked immediately.



Repository ID	Meta Model	Adapter	Resources	Timestamp	Info
re_attribution	re_metamodel	EcoreAttributionAda...	/JWT-EXAMPLES/runninge...	06:36:45:433	<runningexample2.attrm...

Parameter	Value
inheritance_policy	level2
java_rule_classes	
mock_javarules	false
static_rule_creation	true

Repository ID: re_attribution
Meta model ID: re_metamodel

Resources

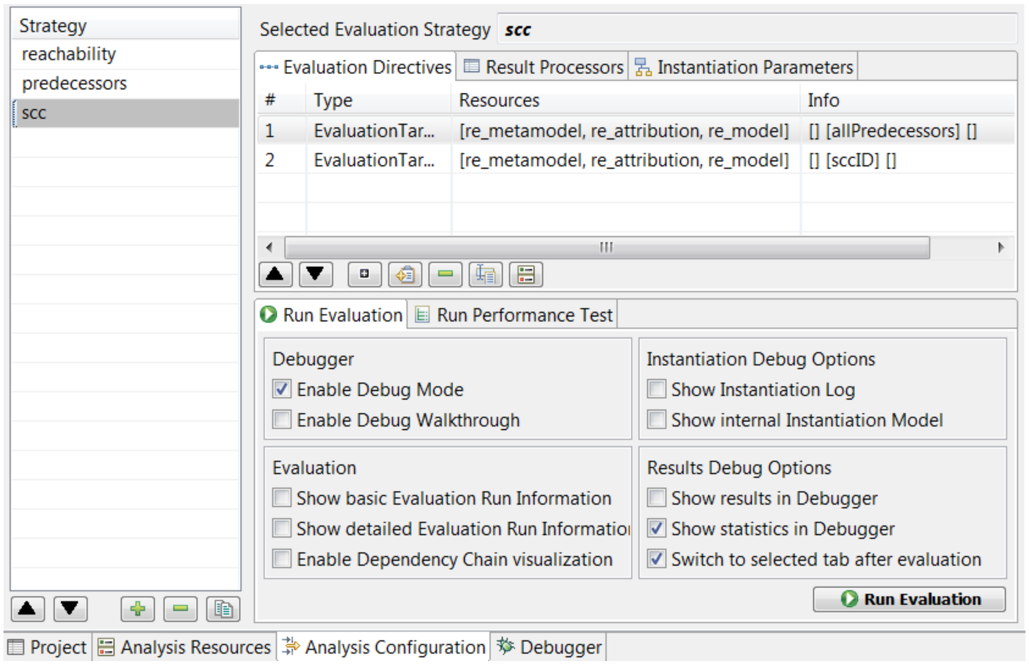
File
runningexample2.attrmm

Figure 8.11.: Resource management in the Project Set Editor.

Figure 8.11 shows the contents of the Analysis Resources tab of the editor. The interface is divided into four categories that support the specification of metamodel, model and attribution resources as well as the definition of loggers and Result Processors. In this screenshot, the attribution section has been selected. The currently active edit dialog displays the properties of the attribution from the running example which has been assigned the repository id re_attribution. The graphical interface provides access to all relevant options for the management of source artifacts, Java rule implementations, the configuration of the AttributionParameter settings and the selection of the corresponding metamodel resource entry (re_metamodel). Additional information about the loaded attribution resources, such as available attributes and attribute inheritance status, is extracted using MAF Core's API functions and is displayed at the bottom of the main window. The pages for the other three types of resources are structured in a similar fashion.

8.3.3. Defining Evaluation Strategies

Once the required resources have been defined, the specification process can continue with the declaration of Evaluation Strategies.



Strategy
reachability
predecessors
scc

Selected Evaluation Strategy **scc**

Evaluation Directives | Result Processors | Instantiation Parameters

#	Type	Resources	Info
1	EvaluationTar...	[re_metamodel, re_attribution, re_model]	[] [allPredecessors] []
2	EvaluationTar...	[re_metamodel, re_attribution, re_model]	[] [sccID] []

Run Evaluation | Run Performance Test

Debugger
- [✓] Enable Debug Mode
- [ ] Enable Debug Walkthrough

Instantiation Debug Options
- [ ] Show Instantiation Log
- [ ] Show internal Instantiation Model

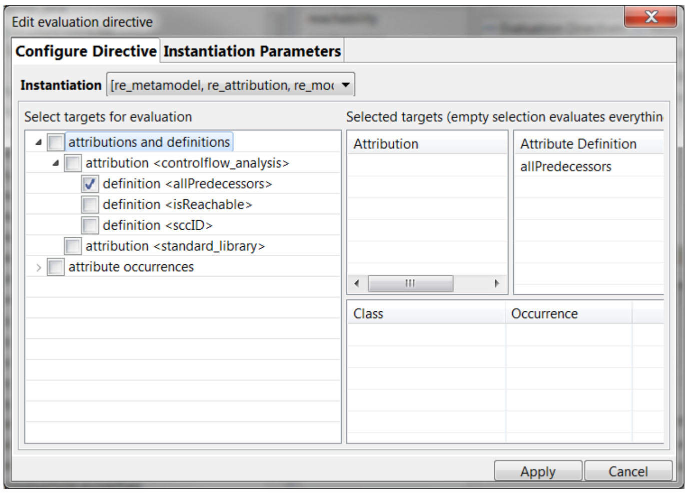
Evaluation
- [ ] Show basic Evaluation Run Information
- [ ] Show detailed Evaluation Run Informatio
- [ ] Enable Dependency Chain visualization

Results Debug Options
- [ ] Show results in Debugger
- [✓] Show statistics in Debugger
- [✓] Switch to selected tab after evaluation

Figure 8.12.: Evaluation Strategies defined for the running example.

The management functions for strategies are located on a separate tab (Analysis Configuration) of the editor. This is depicted in Figure 8.12. On the left hand side, the ids of the already defined strategies are listed. By using the controls at the bottom of this section, it is possible to add, remove, copy or rearrange the strategies. For the running example, three strategies have been created that are responsible for executing the reachability, the predecessor and the SCC analysis respectively.

The list of directives which are part of the currently selected strategy is displayed on the right along with controls that allow the selection of loaded Result Processors and the configuration of InstantiationParameters. Using the dialog shown in Figure 8.13, it is possible to connect a target to one of the available Instantiations and to set up the Attribute Instance Selection. In this dialog, one can also declare target-specific InstantiationParameters that override the strategy's global parameter set. For the strategy scc in the running example (shown in the screenshot), two



Edit evaluation directive

Configure Directive | Instantiation Parameters

Instantiation [re_metamodel, re_attribution, re_mo

Select targets for evaluation

- [ ] attributions and definitions
  - [ ] attribution <controlflow_analysis>
    - [✓] definition <allPredecessors>
    - [ ] definition <isReachable>
    - [ ] definition <sccID>
  - [ ] attribution <standard_library>
- [ ] attribute occurrences

Selected targets (empty selection evaluates everythin

Attribution	Attribute Definition
	allPredecessors

Class	Occurrence

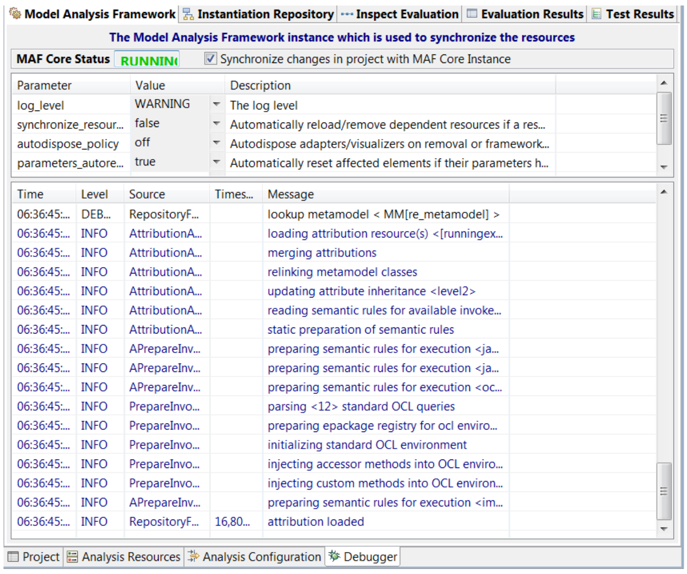
Apply | Cancel

Figure 8.13.: Dialog for configuring the instance selection of an Evaluation Target.

Evaluation Targets have been defined. The first one triggers the computation of all instances of the allPredecessors attribute while the second one will do the same for sccID in a subsequent step.

8.3.4. Analysis Debugging

To support the development of analysis projects, the Project Set Editor includes debugging capabilities. For this purpose, the editor synchronizes the currently loaded Project Set with an instance of MAF Core. Information about the internal status of the project is reported through a registered debug logger which generates a detailed visualization for all evaluation steps.



Model Analysis Framework | Instantiation Repository | Inspect Evaluation | Evaluation Results | Test Results

The Model Analysis Framework instance which is used to synchronize the resources

MAF Core Status RUNNIN([✓] Synchronize changes in project with MAF Core Instance

Parameter	Value	Description
log_level	WARNING	The log level
synchronize_resour...	false	Automatically reload/remove dependent resources if a res...
autodispose_policy	off	Autodispose adapters/visualizers on removal or framework...
parameters_autore...	true	Automatically reset affected elements if their parameters h...

Time	Level	Source	Times...	Message
06:36:45:...	DEB...	RepositoryF...		lookup metamodel < MM[re_metamodel] >
06:36:45:...	INFO	AttributionA...		loading attribution resource(s) <[runningex...
06:36:45:...	INFO	AttributionA...		merging attributions
06:36:45:...	INFO	AttributionA...		relinking metamodel classes
06:36:45:...	INFO	AttributionA...		updating attribute inheritance <level2>
06:36:45:...	INFO	AttributionA...		reading semantic rules for available invoke...
06:36:45:...	INFO	AttributionA...		static preparation of semantic rules
06:36:45:...	INFO	APrepareInv...		preparing semantic rules for execution <ja...
06:36:45:...	INFO	APrepareInv...		preparing semantic rules for execution <ja...
06:36:45:...	INFO	APrepareInv...		preparing semantic rules for execution <oc...
06:36:45:...	INFO	PrepareInvo...		parsing <12> standard OCL queries
06:36:45:...	INFO	PrepareInvo...		preparing epackage registry for ocl enviro...
06:36:45:...	INFO	PrepareInvo...		initializing standard OCL environment
06:36:45:...	INFO	PrepareInvo...		injecting accessor methods into OCL enviro...
06:36:45:...	INFO	PrepareInvo...		injecting custom methods into OCL environ...
06:36:45:...	INFO	APrepareInv...		preparing semantic rules for execution <im...
06:36:45:...	INFO	RepositoryF...	16,80...	attribution loaded

Project | Analysis Resources | Analysis Configuration | Debugger

Figure 8.14.: Running example: MAF core output in debugger.

The debugging facilities can be accessed via the Debugger tab. The first page -

shown in Figure 8.14 - displays the current status of the MAF Core instance and can enable/disable the automatic synchronization with the editor. Additionally, it visualizes the CoreParameters that are used to initialize the evaluator. Finally, all log messages are recorded and shown in a tabular view.

The second tab - Instantiation Repository - provides an overview of all Instantiations which are available in the repository and enables an inspection of their contents as well as monitoring the attribute instantiation process itself.

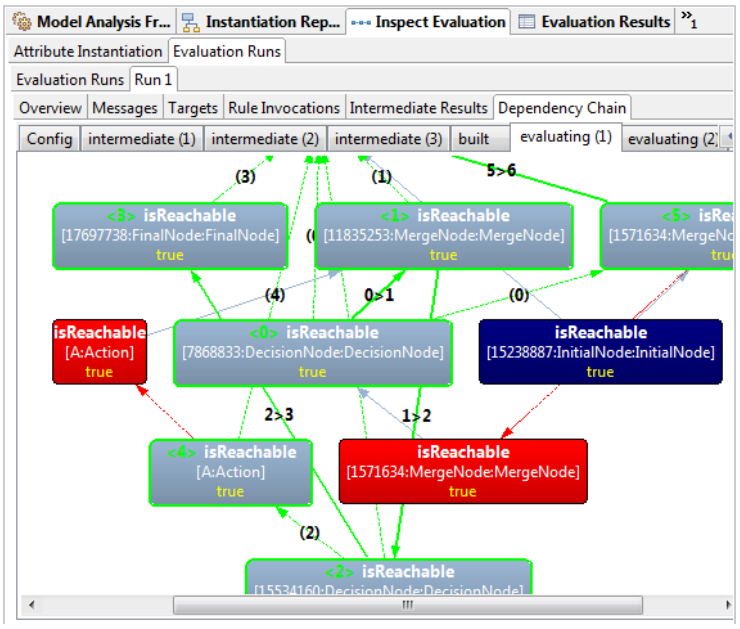

Model Analysis Fr...
Instantiation Rep...
Inspect Evaluation
Evaluation Results
Attribute Instantiation
Evaluation Runs
Evaluation Runs
Run 1
Overview
Messages
Targets
Rule Invocations
Intermediate Results
Dependency Chain
Config
intermediate (1)
intermediate (2)
intermediate (3)
built
evaluating (1)
evaluating (2)
(3)
(1)
5>6
<3> isReachable [17697738:FinalNode:FinalNode] true
<1> isReachable [11835253:MergeNode:MergeNode] true
<5> isRe [1571634:MergeNo tru
(4)
0>1
(0)
isReachable [A:Action] true
<0> isReachable [7868833:DecisionNode:DecisionNode] true
isReachable [15238887:InitialNode:InitialNode] true
2>3
1>2
<4> isReachable [A:Action] true
isReachable [1571634:MergeNode:MergeNode] true
(2)
<2> isReachable [15534160:DecisionNode:DecisionNode]

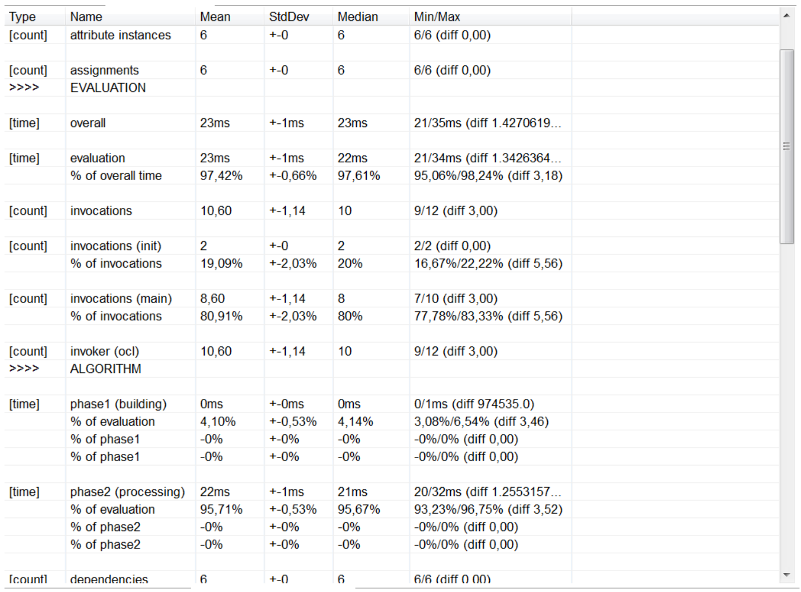
Figure 8.15.: Running example: Different states of the reachability dependency chain visualized in the debugger.

The third page, titled Inspect Evaluation, enables an in-depth examination of the whole evaluation process. The execution of an Evaluation Strategy can be triggered via the Analysis Configuration view as shown in Figure 8.12. In this dialog, several options can be set which influence the evaluation and its visualization in the debugger interface. It is, for example, possible to deactivate the recording of specific data in order to increase the performance.

For the execution of the current strategy, Inspect Evaluation provides detailed information about all steps carried out by the Strategy Executor, the Instantiation and the selected DFA solving algorithm: Evaluation Runs shows an overview of all executed Evaluation Directives while the concrete steps are listed on additional tabs which are generated dynamically for each target. In this example, only one tab (Run 1) is displayed since the executed reachability strategy contains only one target. Information that is collected for this target includes a dedicated log section containing the messages generated by the Data-flow Solver and a list of the requested Attribute Instances according to the target's instance selection properties. It also records the Rule Invocations in the order in which they have been executed along with the instances for which they were invoked and the respective result values. This page also displays statistical information about the amount of rule invocations.

If the dependency chain algorithms are used for DFA solving, some additional information is provided: The Intermediate Results page lists the instance results after each bottom-up processing. The Dependency Chain tab shown in the screenshot in Figure 8.15 is used to visualize the evolvement of the dependency structure during the execution of the selected target. Dependency chains that result from the application of the dependency discovery step are prefixed with intermediate while chains that are currently being processed are labelled as evaluating. The graph contains the attribute instances as colored nodes: Gray for inner nodes, blue for leaf nodes and red for virtual nodes. The recorded output dependencies between the instances are visualized as edges. Nodes and edges which are highlighted in green have been processed in the currently selected iteration with the edge labelings indicating the order in which the solver traversed the chain.

The next main component of the debugger displays the actual Evaluation Results arranged into the three categories of Attribute Assignments and passed/violated Attribute Constraints.



Type	Name	Mean	StdDev	Median	Min/Max
[count]	attribute instances	6	+-0	6	6/6 (diff 0,00)
[count]	assignments	6	+-0	6	6/6 (diff 0,00)
>>>>	EVALUATION				
[time]	overall	23ms	+-1ms	23ms	21/35ms (diff 1.4270619...
[time]	evaluation	23ms	+-1ms	22ms	21/34ms (diff 1.3426364...
	% of overall time	97,42%	+-0,66%	97,61%	95,06%/98,24% (diff 3,18)
[count]	invocations	10,60	+-1,14	10	9/12 (diff 3,00)
[count]	invocations (init)	2	+-0	2	2/2 (diff 0,00)
	% of invocations	19,09%	+-2,03%	20%	16,67%/22,22% (diff 5,56)
[count]	invocations (main)	8,60	+-1,14	8	7/10 (diff 3,00)
	% of invocations	80,91%	+-2,03%	80%	77,78%/83,33% (diff 5,56)
[count]	invoker (ocl)	10,60	+-1,14	10	9/12 (diff 3,00)
>>>>	ALGORITHM				
[time]	phase1 (building)	0ms	+-0ms	0ms	0/1ms (diff 974535.0)
	% of evaluation	4,10%	+-0,53%	4,14%	3,08%/6,54% (diff 3,46)
	% of phase1	-0%	+-0%	-0%	-0%/0% (diff 0,00)
	% of phase1	-0%	+-0%	-0%	-0%/0% (diff 0,00)
[time]	phase2 (processing)	22ms	+-1ms	21ms	20/32ms (diff 1.2553157...
	% of evaluation	95,71%	+-0,53%	95,67%	93,23%/96,75% (diff 3,52)
	% of phase2	-0%	+-0%	-0%	-0%/0% (diff 0,00)
	% of phase2	-0%	+-0%	-0%	-0%/0% (diff 0,00)
[count]	dependencies	6	+-0	6	6/6 (diff 0,00)

Figure 8.16.: Running example: Statistical evaluation of a test run.

The last section is reserved for the integrated testing and statistical evaluation facilities. The option of running a strategy in test mode is also selectable from the menu shown in Figure 8.12. If this mode is activated, the Strategy Executor will execute the strategy repeatedly a given amount of times. After each run, the results and gathered statistical information about the execution are stored and the respective Instantiations are reset to their initial state. Before the next iteration starts, the results are compared against the results of the previous run and deviations are immediately reported. If this is the case, then it can be concluded that the analysis specification is incorrect as it does not converge in a unique fixed-point (i.e. it is indeterministic). Otherwise, if the validation succeeds, the statistical information is averaged and reported in a table as shown in Figure 8.16. This structure provides information about different aspects of the evaluation such as the duration of different steps of the solving process and the number of rule executions in different phases. It lists the overall execution time for the strategy including the instantiation process, the duration of the actual result computation as well as recorded times for different aspects of the solving algorithms such as building and traversing chains and recursive lookup. Additional data includes the number of bottom-up runs, chain merges, discovered dependencies, the amount of different node types etc. For each result, the mean value is displayed alongside the standard deviation, the median and the respective minimal and maximal values.

In conclusion, the debugger provides a comprehensive set of tools that enable developers to test analyses during the specification process and evaluate their performance as well as the correctness of the specifications.

8.3.5. Project Set DSL Editor

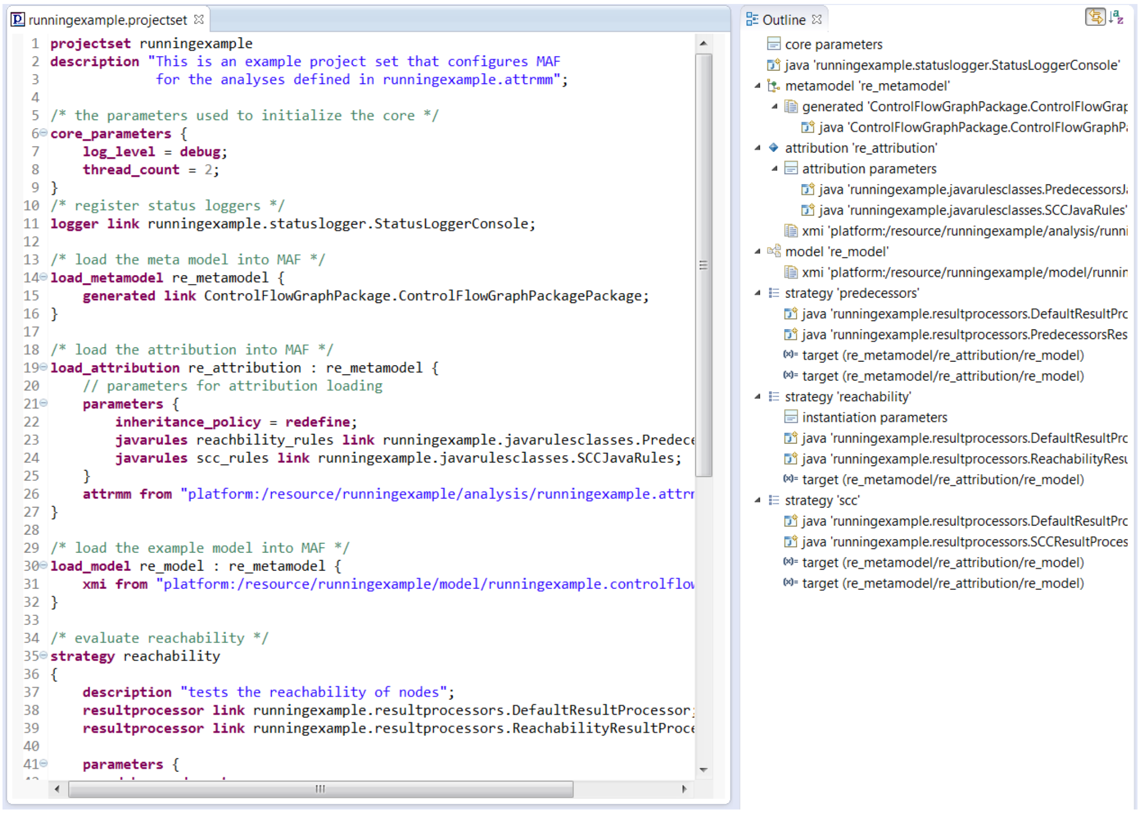
The Project Set Editor described in the previous sections is a useful tool which provides a powerful set of features for editing and testing analyses. Nevertheless, in some cases, a more lightweight approach to the definition of a Project Set configuration might be advantageous. For this purpose, a textual DSL has been developed using the Xtext language workbench. The textual format allows experienced users who are familiar with the structural design of Project Sets to very quickly devise the corresponding definitions by avoiding the encumbrance of wizards and other GUI elements intended to help the more inexperienced user. It also provides a full overview of a definition at a single glance, another feature which can be very beneficial to the experienced developer as it supports an easy implemention of changes which affect multiple elements using basic text editing functionality such as search and replace.

Figure 8.17 shows the textual description of the running example's Project Set in the generated editor. The syntax follows the grammar definition of the DSL which is included in Appendix D.9.

When using the DSL editor, the controls for running an analysis can be found in the toolbar of the Eclipse environment. This is shown in Figure 8.18: If the editor is active, several new options become available in the toolbar. For example, all strategies from the current Project Set definition are listed in a dropdown menu from which they can be directly invoked. The integrated debugger from the Project Set editor is replaced by a window that can be enabled/disabled by toggling the respective control. If the debug window is active, the strategy is automatically executed in debug mode so that all outputs of the framework are captured and visualized. Finally, a separate toggle switch activates the statistical evaluation in which case the strategy is run in the test mode and the collected information is displayed.

8.4. Future Developments

In this chapter, we described the internal design of the Model Analysis Framework. In its current state, this framework already represents a fully-featured solution for the implementation of data-flow based model analyses and the integration of analysis


runningexample.projectset

```
projectset runningexample
description "This is an example project set that configures MAF
             for the analyses defined in runningexample.attrmm";

/* the parameters used to initialize the core */
core_parameters {
    log_level = debug;
    thread_count = 2;
}
/* register status loggers */
logger link runningexample.statuslogger.StatusLoggerConsole;

/* load the meta model into MAF */
load_metamodel re_metamodel {
    generated link ControlFlowGraphPackage.ControlFlowGraphPackagePackage;
}

/* load the attribution into MAF */
load_attribution re_attribution : re_metamodel {
    // parameters for attribution loading
    parameters {
        inheritance_policy = redefine;
        javarules reachbility_rules link runningexample.javarulesclasses.Predece
        javarules scc_rules link runningexample.javarulesclasses.SCCJavaRules;
    }
    attrmm from "platform:/resource/runningexample/analysis/runningexample.attrr
}

/* load the example model into MAF */
load_model re_model : re_metamodel {
    xmi from "platform:/resource/runningexample/model/runningexample.controlflow
}

/* evaluate reachability */
strategy reachability
{
    description "tests the reachability of nodes";
    resultprocessor link runningexample.resultprocessors.DefaultResultProcessor;
    resultprocessor link runningexample.resultprocessors.ReachabilityResultProce

    parameters {
```

Outline

```
core parameters
java 'runningexample.statuslogger.StatusLoggerConsole'
metamodel 're_metamodel'
    generated 'ControlFlowGraphPackage.ControlFlowGrap
        java 'ControlFlowGraphPackage.ControlFlowGraphP
attribution 're_attribution'
    attribution parameters
        java 'runningexample.javarulesclasses.PredecessorsJ
        java 'runningexample.javarulesclasses.SCCJavaRules'
    xmi 'platform:/resource/runningexample/analysis/runni
model 're_model'
    xmi 'platform:/resource/runningexample/model/runnir
strategy 'predecessors'
    java 'runningexample.resultprocessors.DefaultResultPrc
    java 'runningexample.resultprocessors.PredecessorsRes
    target (re_metamodel/re_attribution/re_model)
    target (re_metamodel/re_attribution/re_model)
strategy 'reachability'
    instantiation parameters
    java 'runningexample.resultprocessors.DefaultResultPrc
    java 'runningexample.resultprocessors.ReachabilityResu
    target (re_metamodel/re_attribution/re_model)
strategy 'scc'
    java 'runningexample.resultprocessors.DefaultResultPrc
    java 'runningexample.resultprocessors.SCCResultProces
    target (re_metamodel/re_attribution/re_model)
    target (re_metamodel/re_attribution/re_model)
```

Figure 8.17.: Running example: Project Set specified using a dedicated DSL.

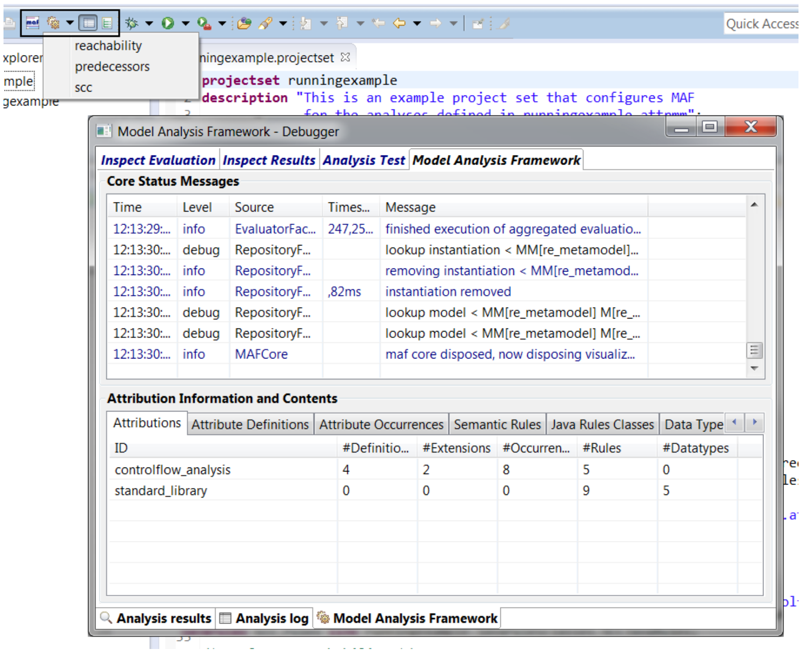
capabilities into third-party applications. For this purpose, MAF has been designed to be highly customizable so that its functionality can be extended to meet the requirements of different technical application domains. Using Eclipse's model-based development facilities, we realized an IDE that simplifies the specification process.

By applying the framework to case studies in different domains (cf. Chapter 10), we were able to identify several points where the existing implementation can be extended to further improve the applicability and versatility of the technique:

Standalone Usage

Currently, the central part of the framework (MAF Core) can be run in standalone mode and thus be included in non-Eclipse based Java applications. Depending on the properties of the target environment, it can however still be a challenge to access the analysis facilities from other programs. For example, the widely-used modeling tool Enterprise Architect[5] expects that plugins are provided in a native binary format. In this case, a Technology Bridge (cf. Figure 8.10) between the different systems must be provided. While it would be possible to implement the required functionality from scratch for each target application, a more generalized approach would be beneficial. More specifically, this could be realized by embedding MAF Core in a server container that exposes the framework's API through a TCP/IP interface instead of Java

[5] http://www.sparxsystems.com.au/



```
projectset runningexample
description "This is an example project set that configures MAF
```

Core Status Messages

Time	Level	Source	Times...	Message
12:13:29:...	info	EvaluatorFac...	247,25...	finished execution of aggregated evaluatio...
12:13:30:...	debug	RepositoryF...		lookup instantiation < MM[re_metamodel]...
12:13:30:...	info	RepositoryF...		removing instantiation < MM[re_metamod...
12:13:30:...	info	RepositoryF...	,82ms	instantiation removed
12:13:30:...	debug	RepositoryF...		lookup model < MM[re_metamodel] M[re_...
12:13:30:...	debug	RepositoryF...		lookup model < MM[re_metamodel] M[re_...
12:13:30:...	info	MAFCore		maf core disposed, now disposing visualiz...

Attribution Information and Contents

Attributions | Attribute Definitions | Attribute Occurrences | Semantic Rules | Java Rules Classes | Data Type

ID	#Definitio...	#Extensions	#Occurren...	#Rules	#Datatypes
controlflow_analysis	4	2	8	5	0
standard_library	0	0	0	9	5

Figure 8.18.: Running example: Project Set DSL editor controls.

method invocations. This would allow developers to access the necessary functions through standardized networking technologies available in almost any relevant environment. To support this feature, a protocol would have to be developed that wraps the API functions of MAF. A prototype - called MAF-Analyzer - which implements parts of this functionality is available from the source code repository and has been employed in the EAM case study (cf. Section 10.2).

Multi-Threading Support

In its current implementation, the framework provides multi-threading support on an analysis level. The computation of analysis results therefore benefits from the availability of multicore hardware. However, this concept does not extend to other parts of the framework. The possibility to carry out multiple actions such as loading models or running analyses concurrently would enable applications to invoke the framework's functions in a non-blocking way. From the user interface's perspective the application would therefore become more responsive. Performance-wise, the evaluation of loaded resources - a task that relies heavily on computing power - and the loading of new source artifacts from the hard drive could be parallelized. On a computer with many processor cores, this feature can also be used to execute multiple independent analyses in parallel. One way to implement this functionality would be to wrap all API calls to MAF Core in parallelizable commands and use MAF's existing work queue infrastructure to submit these tasks for execution.

Multi-User support

The concepts described in the previous two items can be combined and extended to provide a server-based multi-user platform. In this case, a server that encapsulates the functionality of a MAF Core instance and makes the methods available through a network interface could run on a dedicated machine. This becomes relevant if more complex analyses have to be carried out or if the input models are very large so that users would benefit from running the analyses on centralized and more powerful hardware. Server functionality can also include resource management capabilities so that resources can be stored and manipulated on the central machine. This could, for example, be realized by employing EMF persistence frameworks such as the Eclipse projects Teneo[6] or CDO[7]. Coupling the resource management and the analysis functions would provide a single entry point for the management and analysis of models. This would enable applications to automatically invoke the evaluation if models are modified and to report detected problems to all users that are linked to the respective resources. This concept is therefore comparable to a continuous integration server. Additionally, a centralized rights management system could be provided that restricts access to certain resources based on the users' access privileges. On a technological level, this could, for example, be realized using a model-based integration platform such as ModelBus[8].

Integration with the EMF Validation Framework
The EMF Validation Framework[9] is a component of the Eclipse modeling project. It provides an interface for extending models with constraint definitions that can be validated either in live mode (each time a model is modified) or as batch jobs. These validation hooks can be used as entry points for the execution of flow analyses. The validation framework also provides an API for the traversal of models so that partitions that are affected by modifications can be identified. This can be useful when determining the starting points for an analysis so that only changed elements are reevaluated. Finally, the provided facilities can be used to report and visualize detected problems in the Eclipse IDE.

Improvement of Core Functionality
There are several possibilities for the improvement of the core functions of the Model Analysis Framework. For example, it would be possible to extend the Instantiation component with the ability to detect non-converging analyses. An indication for this property is the repeated occurrence of the same results at the leaves of the dependency chain structure. This problem can occur if the specification is incorrect and the result computation does not terminate in a fixed-point solution. Currently, this would result in an infinite loop unless a limit has been specified for the maximal amount of rule executions.

[6]http://www.eclipse.org/modeling/emft/?project=teneo
[7]http://www.eclipse.org/cdo
[8]http://www.modelbus.org/modelbus
[9]http://www.eclipse.org/modeling/emf/?project=validation

Part IV.

Applications and Evaluation

9. Application Scenarios and Analysis Templates

In Section 9.1, we will discuss typical application areas for the data-flow analysis approach and subsequently present an extensible standard library consisting of templates for the implementation of common analyses in Section 9.2.

9.1. Application Scenarios

This section lists recurring application scenarios in the modeling area in which the flow-based model analysis approach can be put to use. The shared characteristic of these scenarios is that they do not depend on a certain implementation technology or are tied to a specific use case or domain but rather represent conceptual approaches to model analysis. Consequently, they apply - with slight modifications - to a wide variety of different application domains. In some cases, complex usage scenarios may even require the combination of multiple methodologies.

In the following, we will present a list of typical application scenarios. For each case, a short motivation is given and the use of data-flow analysis for achieving the desired result is discussed.

Static Model Validation

Model constraint languages such as OCL or EVL (cf. Section 3.1) are typically employed for a static validation of the correctness of the structural composition of models. To this end, they rely on specifications which extend the target language's syntax on the meta layer. In many cases, static validation is necessary because the expressiveness of the syntactical features of a given modeling framework may not be sufficient to encode all restrictions which have to hold for derived models. The extension of the respective language definition, i.e. the metamodel, enables analyses to rely on the syntactic structure of the target modeling language and ensures that they can be automatically executed for all derived instances. Methods which are specifically adapted to the modeling domain make use of common modeling semantics to guide the instantiation process and for providing access to the respective model's features. Consequently, the constraints can retrieve data (such as the values of class properties) from the models and check whether the elements fulfill the stated requirements. Furthermore, analyses may take the immediate environment of the respective elements, such as properties of neighboring objects, into account. This can usually be achieved using a number of inbuilt navigation statements for addressing and evaluating these properties.

The proposed data-flow approach incorporates the ability to employ traditional constraint languages such as OCL or EVL for the specification of analyses. It additionally implements the principles of information propagation and fixed-point evaluation which extend the feature set of the traditional methods. Propagation simplifies the access to information which originates at distant elements, e.g. by exchanging data between nodes in a control-flow graph which are multiple steps apart. It is furthermore capable of incorporating the model's structure in the computation of the results, for example by aggregating or intersecting incoming result sets at gateway nodes. When combined with the fixed-point evaluation semantics, this feature thereby supports an approximation of runtime behavior. In conclusion, the flow analysis approach extends traditional constraint languages with the ability to base the validation process on a wider range of structural and behavioral properties.

Concrete examples for this application scenario can be found in the case studies described in Sections 10.1 and 10.3. In these instances, the developed approach implements consistency checks for business processes and models which encode information about natural language expressions. The JWT case study furthermore demonstrates the realization of a complex validation scenario based on the propagation of tokens (*tokenflow analysis*).

Information Extraction

Traditional validation scenarios are only concerned with the identification of erroneous structures in models. Effectively, the result of a validation is a binary assessment which indicates whether the examined structure conforms to the stated restrictions. However, the usage of analysis techniques also provides the possibility to extract more sophisticated semantic properties. The generated output can be a valuable source of information or may form the input for subsequent algorithms. The data-flow analysis technique provides a powerful tool for accomplishing this goal as it enables the aggregation of flow-sensitive information. This feature can therefore be used to assess properties that require a context-based evaluation of model elements.

While, in theory, this principle can be applied to any modeling language, the potential of this method is perhaps most obvious in the case of behavioral models which encode control-flows. For this type of model, the state of an element can be determined with respect to its position in the overall control-flow structure. It should be noted that, depending on the derived properties and the intended use case, the extraction of (flow-sensitive) information can overlap with other scenarios described in subsequent items.

Multiple examples which represent realizations of the extraction scenario can be found in Section 10.1.2. This use case demonstrates how different control-flow properties can be computed in the BPM domain. Section 10.1.3 additionally explores how the allocation of resources in business processes can be examined while the analyses described in Section 10.4.2 rely on the extraction of control-flow data to support the derivation of valid schedulings for automotive systems. Finally, Section 10.2 applies flow analysis to compute paths in structural models which, in turn, form the input for a calculation of EAM metrics.

Modeling Guidelines / Model Metrics / KPI

The derivation of metrics or key performance indicators (KPIs) represents another application scenario for the proposed approach. This also includes the evaluation of codified modeling guidelines which, in contrast to a structural validation, can be used to quantify the quality of a model on a continuous scale. It can be argued that the computation of metrics can be regarded as a specialized case of the extraction of flow-sensitive information which has been discussed in the previous point.

The benefit of the DFA approach for the calculation of model metrics stems from the fact that the incorporation of flow-sensitive properties can greatly simplify the implementation of more complex algorithms. Evidence for this claim can, for example, be found in Section 10.2.2 where different KPIs from the canonical literature of the EAM domain were (re)implemented using the proposed method.

Through an approximation of dynamic behavior, flow analysis also supports the realization of approaches which rely on an assessment of sophisticated static properties. For example, it is possible to envision a use case in the domain of model-based testing where different coverage criteria are approximated statically to determine whether the modeled behavior is likely to yield suitable test cases. This can be implemented by using DFA to compute all unique traversal paths for a test model and to subsequently calculate a percentage for the contained actions and requirements that indicates the relative amount of paths which will cover these items. The resulting information could be used to drive the generation of small but robust test suites which guarantee that all relevant parts of the target system are considered.

Model Refinement

The scenarios of *model validation* and *information extraction* deal with the assessment of the state of a model and the computation of information which can serve as input for algorithms which are executed in subsequent steps. Alternatively, the information could also be used to refine the model itself. Based on an evaluation of behavioral properties, it is possible to generate advisories that indicate weak spots in the modeled behavior. The user can then decide to apply automatically derived optimizations and subsequently repeat the analysis to iteratively refine the model.

The use case detailed in Section 10.4 implements an iterative refinement scenario in which the structural and behavioral aspects of automotive software systems are analyzed. The results of the respective analyses not only indicate erroneous specifications but are also used to devise a set of problem resolutions which can be iteratively applied to the modeled system to resolve unclear situations and to increase its level of detail.

Declarative Specifications

One advantage of the flow-based approach to model analysis lies in the inherent declarative nature of its specifications. Since the underlying DFA solver is responsible for detecting and managing the dependencies between data-flow attributes and for controlling the fixed-point evaluation process, no imperative evaluation logic for algorithms must be provided. This relieves the developer from being concerned with implementation-specific details.

Furthermore, it can be observed that many algorithms with existing imperative implementations depend on flow-sensitive information or naturally incorporate the principle of information propagation to compute contextual results. As a consequence, one can expect that a flow-based implementation of such an algorithm would be easier to realize and more intuitive to understand and to extend than an imperative solution.

This assumption is evaluated in the case studies. For this purpose, we developed flow-based analyses which mirror the functionality of several imperative algorithms. One example consists of the tokenflow method described in Section 10.1.4 which propagates tokens in control-flow graphs using petrinet-like semantics to detect structural components. Another example can be found in the clone detection use case (cf. Section 10.1.5). Furthermore, this scenario also applies for the use case in Section 10.2 which computes various metrics and change impacts for the EAM domain.

Model Transformations

Any analysis technique whose specifications rely on the structural composition of the respective target language can also be employed to implement transformation algorithms. An example for this principle in the domain of compiler construction can be found in [Aho+06]. In this case, so-called *syntax-directed translations* are used to convert language expressions between different representations. More specifically, the authors describe how attribute grammars are able to output translated versions of the original expressions. Since the evaluation of the respective attribution follows the syntactical structure of the language, the transformation process can easily preserve the correct nesting of (sub)expressions and additionally incorporate contextual information during the translation of each construct. This approach therefore enables a declarative specification of transformations without the need for an additional framework.

It can be assumed that this approach is also applicable in the modeling domain. For this purpose, DFA rules could be extended to output a translated expression based on the context of each attribute instance. Effectively, this approach would therefore constitute a methodology for the realization of model transformations. Possible use cases include the automatic generation of documentations or executable implementations from model data.

Programming Languages

In some cases, modeling languages encode exhaustive information which relates to the execution of the modeled behavior. Examples include models in the area of model-based testing which are often structured similarly to UML State Machines or Activity Diagrams. Since these diagrams are used to derive test paths, they usually contain information for controlling the test execution and for validating the system's behavior. For this purpose, the elements of test models are often annotated with program instructions that configure the system or with guards that check whether certain conditions are met. It is obvious that this format bears some resemblance to the program graphs which form the input for traditional data-flow analysis. It can therefore be assumed that some of the typical analyses found in the field of compiler construction can be adapted for use in this domain. A *reaching definitions* analysis, for example, could be used to track the usage of variables in test models. This knowledge can be of use during the design phase as it provides valuable insights into the behavior of the modeled system such as paths which may not be executable due to conflicting specifications.

Another potential use case can be found in the GeCoS compiler suite [Der+] which implements a model-based representation for the abstract syntax trees of C programs. This approach has the advantage of supporting the direct application of traditional modeling techniques such as model transformations to program code. The application of the data-flow analysis technique in this scenario is straightforward as it exactly mirrors its usage in compilers.

9.2. Analysis Templates

In this section, we will present a variety of templates which demonstrate how the data-flow analysis approach can be employed for the realization of very generic goals. Consequently, many of these analyses can be adapted to different domains and modified to compute different kinds of results. The presented templates therefore constitute building blocks for custom analysis functions. This listing is by no means an exhaustive compilation of all possible use cases for the DFA methodology. Rather, it is intended as an inspiration and a starting point for custom explorations of the possibilities offered by the described method. To specify the executable parts of the analyses, we employ a language which follows the notion of imperative OCL. This choice was motivated by the fact that the navigational statements of this language allow for a short and concise description of the relevant computation steps. Furthermore, the focus on the functional aspects means that the provided templates can be easily adapted to other implementation languages such as Java.

In general, these analyses can be easily adapted to any modeling language which possesses the layout of a directed graph and which implements control-flow semantics. Furthermore, they can also be used to extract a variety of information from structural models, depending on the domain-specific semantics of the target DSL. The analyses are therefore applicable in a large number of different domains and can be employed to achieve different goals.

9.2.1. Reachability/Liveness

A property shared by any directed graph is the *reachability* and *liveness* state of its constituents. An element is *reachable* if there exists a path from the first node to the element. Correspondingly, it is *live* if the final element is reachable.

Algorithm 13 Reachability analysis

```
1: Attribution CFG_REACHABILITY
2:     attribute assignment isReachable : Boolean initWith false
3:     extend StartNode with
4:         occurrenceOf isReachable calculateWith true
5:     extend Node with
6:         occurrenceOf isReachable calculateWith
7:             "return self.incoming.source.isReachable()->includes(true)"
```

Algorithm 13 defines a simple reachability analysis. This specification is a variation of the running example from Chapter 6. It defines an attribute isReachable of type Boolean which is initialized with the value false. The result for isReachable at a Node depends on the reachability state of its immediate predecessors. If at least one predecessor is reachable, the current Node is reachable as well. By default, the StartNode is classified as reachable.

Algorithm 14 Liveness analysis

```
1: Attribution CFG_LIVENESS
2:     attribute assignment isLive : Boolean initWith false
3:     extend EndNode with
4:         occurrenceOf isLive calculateWith true
5:     extend Node with
6:         occurrenceOf isLive calculateWith
7:             "return self.outgoing.target.isLive()->includes(true)"
```

Algorithm 14 provides a modified version of the reachability analysis which determines the liveness state. In this case, the EndNode is considered to be live while all other Nodes are live if this property is true for at least one successor.

The listed algorithms can be easily extended to incorporate other information in the assessment of whether an element is reachable or live. For example, based on an evaluation of guards at model Edges, branches could be excluded which cannot be reached due to conflicting restrictions. Furthermore, it could be stated that in the case of multiple incoming edges, all predecessors (successors) must be reachable (live). This modification is useful if the target modeling language specifies synchronization points, e.g. JoinNodes for parallel paths, which require that all incoming paths must be reachable to continue with the execution.

9.2.2. Flow Sets (Predecessors/Successors)

In Chapter 6, we already briefly explored an approach for the extraction of control-flow data which relates to the predecessor and successor relationships in the model: In a directed graph, each node has a set of transitive predecessor and successor nodes. It is furthermore possible to distinguish between the set of minimal (guaranteed) and maximal (possible) predecessors and successors. As the naming suggests, these results convey information about the dynamic behavior of the system by indicating which nodes *will* or *may* be traversed *before* or *after* the execution reaches the respective node. This knowledge can be useful in many instances as it relays basic information about the properties of the model with respect to the semantics of the target domain. It also forms the input for subsequent analyses such as the SCC detection described in the next section.

Algorithm 15 Predecessor analysis

```
Attribution CFG_PREDECESSORS
    attribute assignment allPredecessorsMin : Set(Node) initWith INIT
    attribute assignment allPredecessorsMax : Set(Node) initWith {}

    extend StartNode with
        occurrenceOf allPredecessorsMin calculateWith {}
    extend Node with
        occurrenceOf allPredecessorsMin calculateWith
            “var result : Set(Node) := null;
             self.incoming.source->forEach(predecessor) {
                var predecessorResult : Set(Node) := predecessor.allPredecessorsMin();
                if (result = null) then
                    result := predecessorResult
                else if (not predecessorResult = INIT and not result = INIT) then
                    result := result->intersection(predecessorResult)
                endif;
             }
             return result”
        occurrenceOf allPredecessorsMax calculateWith
            “return self.source.incoming->union(
                self.source.incoming.allPredecessorsMax())”
```

The attribution in Algorithm 15 outlines the computation of the minimal and maximal predecessor sets for control-flow models. The successor sets can be computed correspondingly by adapting this specification to consider succeeding instead of preceding elements. This principle is evident from the liveness analysis in Algorithm 14 which reverses the direction of the reachability analysis.

The computation of the attribute allPredecessorsMax is straightforward. The result consists of the union of the direct predecessors and their respective transitive predecessor sets which are accessed via allPredecessorsMax(). Consequently, the results yielded by this analysis provide each Node with the set of its transitive predecessors. Alternatively, it would also be possible to include the current element (`self`) in the result set. In this case, each Node would be regarded as a predecessor of itself which can be useful in certain scenarios.

In the presented solution, the computation of the attribute allPredecessorsMin is slightly more complicated as it employs the generic initialization constant INIT instead of the universal set $\mathcal{U}$ as start value for the fixed-point iterations. This approach has the obvious advantage of not requiring any previous knowledge about the nature of the value domain. Consequently, the data-flow rule must be adapted so as to treat INIT as a neutral value. For this purpose, the calculation starts with an empty result set and iteratively intersects this result with the preceding results, provided they already possess a value $\neq$ INIT. If, however, all inputs are initialization values, the INIT marker is preserved to indicate that the current iteration value still conforms to $\top$ and must be recomputed in a later iteration.

Algorithm 16 Predecessor analysis (alternative version)

```
1: Attribution CFG_PREDECESSORS
2:     attribute assignment allPredecessorsMin : Set(Node) initWith
3:         "return value domain"
4:     extend Node with
5:         occurrenceOf allPredecessorsMin calculateWith
6:             "return self.source.incoming->intersection(
7:                 self.source.incoming.allPredecessorsMin())"
```

Conversely, if the value domain is known, the computation of the minimal predecessor set can be carried out similarly to the maximal result. This is demonstrated in the alternative specification for allPredecessorsMin depicted in Algorithm 16. In this case, the initialization rule must return the set of all Nodes. For some modeling languages, this can be achieved by accessing the respective container element and retrieving the set of all available elements.

9.2.3. SCC Detection

A common requirement when analysing control-flow graphs consists of the identification of cyclic structures. Information about the maximal connected subgraphs - or the strongly connected components - is a required input for many other algorithms. This can be achieved, for example, by using Tarjan's algorithm [Tar72]. Alternatively, it is also possible to implement this feature using data-flow analysis. This has the benefit of a very intuitive specification that can be easily adapted to the requirements of the respective target domain. Furthermore, the computed results can be made available as input for subsequent flow analyses which rely on this information.

Algorithm 17 SCC ID analysis

```
Attribution CFG_SCCID
    attribute assignment sccID : Integer initWith -1

    extend Node with
        occurrenceOf sccID calculateWith
            “var predecessors : Set(Node) := self.allPredecessorsMax();
             if (predecessors = INIT) then
                 return -1
             else if (predecessors->includes(self)) then
                 return predecessors->hashCode()
             endif;
             return 0”
```

The attribution presented in Algorithm 17 assigns an identifier sccID to each node that uniquely identifies the detected SCC or has the value 0 if the node does not belong to a cycle. For this purpose, the value for sccID is initially set to -1, indicating that the instance has not yet been processed. Subsequently, the analysis requests the maximal predecessor set as described in the last section. If the predecessors have not yet been fully computed, i.e. if their value is still INIT, the initialization value for the SCC attribute is returned accordingly[1]. Otherwise, it is checked whether the predecessor set contains the context element. If a node is a predecessor of itself, it is part of a cycle and a corresponding identification value is generated from the hash codes of the predecessors[2]. As, by definition, all nodes belonging to the same maximal cycle share the same predecessors, this is a feasible way to generate a unique identification for each cyclic structure.

[1]This is not necessary if one can ensure that the predecessor analysis is completed before the SCC analysis runs.

[2]Alternatively, the transitive predecessors of the current Node could be compared to the union of the allPredecessorsMax results computed for the direct predecessors.

Algorithm 18 SCC objects analysis

```
Attribution CFG_SCCOBJECTS
    attribute assignment sccObjects : Set(Node) initWith INIT

    extend Node with
        occurrenceOf sccObjects calculateWith
            "var sccID : Integer := self.sccID();
             if (sccID = -1) then return INIT
             else if (sccID = 0) then return Set{} endif;
             var result : Set(Node) := Set{self};
             self.incoming.source->forEach(predecessor) {
                 var predecessorResult : Set(Node) := predecessor.sccObjects();
                 if (predecessor.sccID() == sccID and not predecessorResult = INIT)
                     then result := result->union(predecessorResult) endif;
             }
             return result"
```

The identified cycles can be extracted from the analysis results by aggregating all nodes which share the same identifier $\neq 0$ in a group (and elements with sccID $= 0$ representing non-cyclic nodes). However, subsequent analyses may require information about which nodes belong to which cycle as input. For this purpose, it is possible to define an additional analysis, as shown in Algorithm 18, which computes this information. This attribution stores the set of all nodes belonging to the cycle at the respective cyclic nodes.

If the current node belongs to a cycle, i.e. if sccID > 0, then the set of cyclic nodes is aggregated much like in the maximal predecessor analysis. More specifically, the value of the sccObjects attribute at the direct predecessors is queried and - if it is $\neq$ INIT - is added to the result set.

9.2.4. Alternative Paths

In many cases, it can be beneficial to derive specific execution path alternatives for control-flow models. The predecessor/successor analysis yields only general information about elements which may or must be visited on the route to a target object. By considering all alternative routes in the control-flow, it is possible to generate a set of all sequences which lead to a given target element. The information generated by this analysis could then, for example, be used to assess how many path variations exist in a model or how many of the available alternatives fulfill a given set of properties. Since cycles in the control-flow introduce paths of infinite length, it is necessary to implement restrictions which guarantee that the result set is finite. Common conditions may, for example, state that each Node or Edge can only have a limited number of occurrences in each computed sequence.

Algorithm 19 Flow paths analysis

```
 1: Attribution CFG_FLOWPATHS
 2:    attribute assignment flowPaths : Set(List(Element)) initWith {}
 3:    extend StartNode with
 4:       occurrenceOf flowPaths calculateWith
 5:          “return Set{List{self}}”
 6:    extend Node with
 7:       occurrenceOf flowPaths calculateWith
 8:          “var result : Set(List(Element)) := Set{};
 9:           self.incoming->forEach(edge) {
10:              edge.source.flowPaths()->forEach(path) {
11:                 if (not path->contains(self)) then
12:                   result := result->add(List{path}->append(edge)->append(self));
13:                 endif;
14:              }
15:           }
16:           return result”
```

The analysis described in Algorithm 19 realizes this goal by computing an attribute flowPaths for each Node. The result consists of a set of paths (represented as ordered lists) from the StartNode to the respective target. For the first element, a path set with a single entry is created which contains only the StartNode. All other results are computed by iterating over the incoming Edges and checking whether each sequence at the respective source already contains the context element. If this is not the case, a new entry is added to the result collection which appends the incoming Edge as well as the context element to the received path. The inclusion of edges is important to uniquely identify the resulting sequence since there may exist different relationships connecting the same two elements.

The presented analysis may be modified in a number of ways. In its current form, each Node will appear at most once in each of the generated alternative routes. By substituting the check in line [11] with a test that ensures that each Edge may occur only once, Nodes can be “traversed” multiple times, thereby generating a potentially larger result set. It would also be possible to adapt the analysis to incorporate the capability to compute parallel paths.

Since this analysis yields different path variations which conform to the stated restrictions, the results can, for example, be used to aggregate information along the detected sequences and thereby subject the modeled behavior to a statistical evaluation. It is also conceivable that paths derived from Activity Diagrams or State Machines are used as input for a model-based “simulation” of the modeled system.

9.2.5. Definition/Usage Analysis

In some domains, the modeled behavior not only describes the flow of control for runtime instances of the modeled system but also includes specifications about the production/consuming of resources. An example for this application scenario is explored in Section 10.1.3 which examines the availability of data objects which function as the input and output of actions in business process models. An alternative scenario could consist of the declaration and manipulation of program variables inside statements assigned to the nodes and edges of MBT models.

In a *definition/usage* scenario, it is typically assumed that any resource which is read (or manipulated) during the execution of the modeled system must be defined at an earlier point. This principle mirrors the notion of statically typed programming languages which require that any access to a variable is preceded by a declaration statement. A definition/usage analysis which examines the availability status of elements that are produced and consumed at control-flow elements can therefore be beneficial in the task of establishing the correctness of the modeled behavior. An approximation of the runtime behavior based on the computation of the minimal and maximal fixed-point results enables the derivation of different kinds of conclusions about whether the specification is correct for all runtime instances or if there exists at least one execution which may lead to problematic situations.

Algorithm 20 Available resources analysis

```
Attribution DEFUSE_AVAILABLE
   attribute assignment availableResourcesMin : Set(Resource)
      initWith INIT
   extend StartNode with
      occurrenceOf availableResourcesMin calculateWith {}
   extend Node with
      occurrenceOf availableResourcesMin calculateWith
         “var result : Set(Resource) := null;
          self.incoming.source->forEach(predecessor) {
             var resultPred : Set(Resource) := predecessor.availableResourcesMin();
             if (result = null) then
                result := resultPred
             else if (not resultPred = INIT and not result = INIT) then
                if (self.IsKindOf(JoinNode)) then
                   result := result->union(resultPred)
                else
                   result := result->intersection(resultPred)
                endif;
             endif;
          }
          if (not result = INIT) then
             result := result->union(self.availableResources()) endif;
          return result”
```

Algorithm 20 defines an attribution which computes the minimal set of defined resources which are available at the control-flow elements. The result at each Node therefore corresponds to the set of resources for which we can *guarantee* that they will have been defined before this point in the execution flow is reached. Consequently, this set excludes resources produced in branches or inside cycles as not all sequences leading to the target element will traverse these paths. This is accomplished by intersecting the incoming result sets. However, an exception is made for JoinNodes which we assume carry the implicit semantics that all incoming paths must be traversed for the execution to continue. Just like the analysis in Section 9.2.2, this specification is able to handle unspecified value domains by relying on the INIT marker.

The computation of the maximal set of defined resources can be easily implemented by removing the intersection operator and instead unifying the incoming results for all possible Node types. In this case, the analysis yields the set of resources which are *potentially* available at each Node. In other words, the computed results indicate which data objects *may* be available at each control-flow element since at least one route exists on which the respective resource is produced.

Algorithm 21 Unused resources analysis

```
Attribution DEFUSE_UNUSED
    attribute assignment unusedResourcesMin : Set(Resource)
        initWith INIT
    extend StartNode with
        occurrenceOf unusedResourcesMin calculateWith {}
    extend Node with
        occurrenceOf unusedResourcesMin calculateWith
            “var result : Set(Resource) := null;
            self.incoming.source->forEach(predecessor) {
                var resultPred : Set(Resource) := predecessor.unusedResourcesMin();
                if (result = null) then
                    result := resultPred
                else if (not resultPred = INIT and not result = INIT) then
                    if (self.IsKindOf(JoinNode)) then
                        result := result->union(resultPred)
                    else
                        result := result->intersection(resultPred)
                    endif;
                endif;
            }
            if (not result = INIT) then
                result := result->union(self.definedResources())
                result := result->remove(self.usedResources())
            endif;
            return result”
```

It is also possible to compute the “life span” for defined resources, i.e. the partial paths on which resources are available as they have been defined at an earlier point but have not yet been read, modified or consumed. This result can give an indication on how long resources may have to be stored in memory before they can be released but it also enables an identification of write accesses to a specific resource and whether data may be produced but never consumed in subsequent steps.

The attribution shown in Algorithm 21 bears similarities to the analysis from Algorithm 20 as it propagates data-flow results - computed by unusedResourcesMin - along the direction of the control-flow. However, this specification removes (in DFA terms *kills*) locally used resources [23]. Again, the analysis can be adapted to compute the maximal fixed-point approximation.

Algorithm 22 Missing resources analysis

```
Attribution DEFUSE_MISSING
    attribute assignment missingResourcesMin : Set(Resource)
        initWith INIT
    attribute assignment missingResourcesMax : Set(Resource)
        initWith INIT
    extend Node with
        occurrenceOf missingResourcesMin calculateWith
            “var availableResources : Set(Resource) := self.availableResourcesMin();
             if (availableResources = INIT) then return INIT endif;
             var result : Set(Resource) := Set{availableResources};
             return result->remove(self.usedResources())”
        occurrenceOf missingResourcesMax calculateWith
            “var availableResources : Set(Resource) := self.availableResourcesMax();
             if (availableResources = INIT) then return INIT endif;
             var result : Set(Resource) := Set{availableResources};
             return result->remove(self.usedResources())”
```

The results of the analysis of defined resources can be used for different purposes such as a qualitative manual assessment of the properties of the modeled system. Based on the results, it can also be determined whether all of the resources which are locally *used*, i.e. required as input, are available. This allows an automatic validation of the definition/usage relationships. Algorithm 22 demonstrates this process for the minimal and maximal approximations of resource availability. Since the results for missingResourcesMax relies on the potentially available resources, an resource that is identified as missing indicates an error for *all* runtime instances. It is therefore a definitive violation which must be corrected. Conversely, the results for missingResourcesMax lists resources for which an illegal access occurs on *at least one* possible execution. Depending on the semantics of the modeled system, this may or may not equate to an erroneous specification.

The presented algorithms can be extended to provide support for advanced concepts such as an examination of the definition/usage behavior for parallel paths. For example, it would be possible to evaluate if the presence of parallel paths may lead to concurrent accesses to the same resource. Furthermore, if there exists only a limited number of specific resource instances, the read/write accesses could be tracked and the required amount of instances could be approximated for both the best and the worst case scenario.

9.2.6. Type Analysis

Flow analysis can also be used to derive typing information for structural models which encode some type of generalization hierarchy for elements. A variation of this approach has been used in Section 6.3.3 to demonstrate the application of DFA for the computation of the concrete type of the target model elements for attribute instance annotations.

Algorithm 23 Unique root types analysis

```
Attribution TYPE_ROOTTYPES
  attribute assignment rootTypes : Set(Node) initWith {}
  attribute constraint rootTypeUnique : error "no unique root type"
  extend Node with
    occurrenceOf rootTypes calculateWith
      "var result : Set(Node) := Set{};
      if (self.parent = null) then
        result := result->include(self.parent)
      else
        result := result->include(self.parent.rootTypes())
      endif
      return result"
    occurrenceOf rootTypeUnique calculateWith
      "if (self.rootTypes()->size() > 1) then
        self.rootTypeUnique("multiple roots:" +
         self.rootTypes().name + "for" + self.name);
        return false
      endif;
      return true"
```

The specification shown in Algorithm 23 attaches an attribute rootTypes to the Node concept of the target modeling language. In this context, we assume that each Node conforms to a type declaration which may possess a parent type. The analysis determines the respective root(s) for each element which is part of the hierarchy. For this purpose, root entries are added to the result set and propagated to child elements. In many cases, it is required that each element is derived from exactly one unique root. A subsequent evaluation of the rootTypeUnique constraint generates corresponding error messages for all Nodes with multiple root type entries.

Algorithm 24 Test for cyclic hierarchies

```
Attribution TYPE_CYCLES
    attribute constraint rootTypeCycles : error "type hierarchy cyclic"
    extend Node with
        occurrenceOf rootTypeCycles calculateWith
            "if (self.sccID() > 0) then
                self.rootTypeCycles("cyclic hierarchy:" + ;
                 self.sccObjects()).name + "for" + self.name);
                return false
            endif;
            return true"
```

It is also often required that type or generalization hierarchies are acyclic. The SCC detection described in Section 9.2.3 can be used for this purpose. The constraint rootTypeCycles in Algorithm 24 relies on the value of the attribute sccID to identify cyclic paths and to generate an appropriate feedback message.

While the presented analyses address typical problems that may occur in hierarchical type definitions, they can also be applied to different kinds of tree-like structures.

9.2.7. Context-Sensitive Analyses

The analyses described in the last sections rely on the evaluation of contextual information to derive the correct results. However, in practice, it is often the case that the target modeling language provides mechanics for organizing the contents in a containment hierarchy. Examples for this principle include the use of packages in structural models or subprocesses in business processes. To perform a global analysis, i.e. to examine each element in its overall context, it is therefore necessary to propagate the results to nested model artifacts. The relevance of this concept and its application in practice is explored in the case study in Section 10.1.

Algorithm 25 Context-sensitive analysis

```
Attribution SYNTHESIZE_CONTEXT
    attribute assignment attribute : Type initWith Value

    extend StartNode with
        occurrenceOf attribute calculateWith
            "if (self.isKindOf(StartNode) and not self.container = null) then
                return self.container.getAttribute()
             endif;
             return result"

    extend Node with
        occurrenceOf attribute calculateWith
            "self.incoming.source->forEach(predecessor) {
                if (predecessor.isKindOf(Container)) then
                    predecessor.nodes->forEach(containedNode) {
                        if (containedNode.isKindOf(EndNode)) then
                            return containedNode.getAttribute() endif;
                    }
                endif;
             }
             return result"
```

The template shown in Algorithm 25 describes a generic way to address this problem. A data-flow attribute - representing the information which should be evaluated in the model's overall context - is attached to the elements of the target model. The rule for StartNode checks whether the current object resides in a Container. If this is the case, it requests and returns the Container's result. This process propagates data-flow information to the starting point of nested structures such as subprocesses in a business model. Conversely, the results computed inside the substructure must also be relayed back to the surrounding environment. This happens inside the rule for Nodes which query their direct predecessors for Container elements. If a predecessor contains a nested control-flow, the result from the respective EndNode is transfered to the current element. Together, these rules therefore realize the inheritance and the synthesis of data-flow results in nested models.

10. Case Studies and Applications

In this chapter, we will present several case studies which rely on the developed flow analysis technique. Each of the following sections focuses on a single application domain or technological space ranging from areas such as the analysis of business processes to Enterprise Architecture Management. For every domain, we describe several use cases, each of which realizes a specific goal by implementing application scenarios such as model validation or the extraction and processing of semantic properties. In their entirety, the case studies demonstrate how consistent and powerful analysis frameworks can be built for different domains using the method detailed in this thesis. Each case study provides a short introduction of the target domain followed by a description of the goals and subsequently lists the analysis definitions in a pseudo code notation. Each implementation concludes with a discussion of the relevant aspects with respect to the application of the analysis methodology in the context of the respective domain.

It should be noted that the case studies do not focus on an in-depth evaluation of the use cases themselves but rather are intended to highlight the versatility of the approach and demonstrate how complex solutions can be built incrementally on the basis of the application scenarios and the analysis templates from Chapter 9. In short, the following sections prove the applicability of the DFA method for the implementation of analyses in different application areas.

Several factors contribute to the individual challenges which must be overcome to apply the flow analysis method for a specific use case:

Application domain Each of the case studies introduces a new application domain. This diversity illustrates the viability of the analysis technique with respect to its application in arbitrary areas which rely on the modeling paradigm.

Application scenario Analyses may, for example, be used to extract semantic information which is implicitly encoded in the syntactical structure of a model, for the validation of static features of a modeling language or the computation of metrics. More complex use cases may even require the combination of different application scenarios.

Model structure The development of an analysis requires the consideration of the specific properties of the modeling language which is employed to encode information in the respective domain. Modeling languages may focus on the representation of structural or behavioral features or on a combination thereof and can use different approaches to encode these properties.

Incremental analysis A concrete analysis may rely on results which have been computed in previous steps as well as drawing from the DFA standard library in order to iteratively build a comprehensive analysis framework for the given task.

Technical integration Seamless integration on a technical level is an additional challenge that has to be evaluated in the context of realistic applications of the developed technique.

The first case study, which is presented in Section 10.1, implements multiple use cases in the area of business process analysis. Business processes are often represented as directed graphs and therefore provide the opportunity to apply a variety of common DFA methods such as reachability analysis or the computation of transitive successor sets. In addition to these traditional use cases, this case study describes flow-based implementations of existing algorithms for use cases such as SESE decomposition and clone detection. In Section 10.2, we examine possible applications of the DFA technique in the context of EAM and describe how the results can be used to generate comprehensive assessments of the modeled IT landscapes. Section 10.3 deals with the analysis of models which encode semantic information about natural language patterns. Finally, in Section 10.4, we take a look at the AUTOSAR method which is used primarily in the automotive domain. The goal here is to improve existing development methodologies by providing engineers with facilities for the validation and improvement of their system design. While the first case study focuses on the evaluation of behavioral models, the following two sections demonstrate the analysis of structural information. The models in the last case study represent a combination of both paradigms.

10.1. Case Study: Business Process Modeling

In this case study, we focus on the area of business process modeling, which in many organizations has become an integral technology for the management of business processes. For the analysis of process models, we employ both traditional DFA algorithms and devise new, domain-specific techniques. Some of these analyses are reimplementations of existing approaches that have been adapted to follow the declarative paradigm that lies at the core of flow-based analysis. To evaluate the practical aspects of the developed methods, we have integrated the analyses in the BPM tool Java Workflow Tooling (JWT).

10.1.1. Introduction and Motivation

Many competing definitions exist for terms such as business process modeling or (business) process analysis. Since this case study is intended both as a proof-of-concept and an evaluation of the capabilities of DFA in a concrete application domain, we limit ourselves to a very specific interpretation of these concepts which is relevant to the use cases in this section. For additional information about management aspects and different techniques and standards for designing and analyzing business processes, we refer to literature such as [LS07] and [VTM08].

[WFM96] defines a process model as a *"formalized view of a business process, represented as a co-ordinated (parallel and/or serial) set of process activities that are connected in order to achieve a common goal"*. In [FH02], a process model is described as an *"abstract representation of a process architecture, design, or definition"*. It is further stated that *"process models are used where use of the complete process is undesirable or impractical. A process model can be analyzed, validated, and [...] may be used to assist in process analysis, to aid in process understanding, or to predict process behavior"*.

[WFM96] also describes a set of generic concepts supported by many modeling languages in the BPM field. These include control-flow constructs such as sequential, parallel or exclusive paths. Arguably one of the most widely used standards for the modeling of business processes is OMG's graph-oriented Business Process Modeling Notation (BPMN) language which, as the authors of [Woh+06] conclude *"provides direct support for the majority of the control-flow patterns and for nearly half of the data patterns, while support for the resource patterns is scant"*. Control-flow patterns are the most relevant type in the context of this case study as they enable the application of flow-based analysis.

In [VHW03], the importance of process analysis is stressed since *"it is preferable to identify any problems in software before it is actually deployed. In the case of Business Process Models this is especially important as they may involve core business and/or complex business transactions"*. Also, it is stated that analysis can be *"used to investigate ways of improving processes (e.g. reducing their cost)"*.

[Aal07] differentiates between the following types of process analysis:

Validation Testing whether the process behaves as expected.

Verification Establishing the correctness of a process definition.

Performance analysis Evaluating the ability to meet requirements with respect to throughput times, service levels, and resource utilization.

Additionally, an analysis can be classified as being static - i.e. executed during the design time where *"the only basis for analysis is a model"* - or as a dynamic analysis during runtime where *"one can also observe the actual behavior and use this as input for analysis"*.

Data-flow analysis is well-suited for validation purposes as well as for the computation of performance properties as it evaluates processes according to their structural composition by taking into account the context in which objects appear in relation to the overall process. Since data-flow analysis approximates properties that can be derived statically, it constitutes a *design time analysis* operating on the *process model*.

Java Workflow Tooling

The Java Workflow Tooling (JWT) project is part of the yearly official Eclipse release and belongs to the SOA (Service-oriented Architecture) top-level project. Since 2010, the author of this thesis has served as the co-project lead of JWT.

On the official project page[1], the goals of JWT are summarized as follows: *"Eclipse SOA's Java Workflow Tooling project (JWT) provides design time, development time and runtime workflow tools. It also fosters interoperability between Business Process Management (BPM) platforms and integration in Information Systems thanks to Service Oriented Architecture (SOA)."*

Because JWT provides support for the core expressions of the Business Process Modeling Notation[2], we can consider it to be a representative for graph-based business process modeling languages in general.

The relevant property of BPM that makes this area a suitable candidate for studying the application of DFA is the inherent control-flow structure of business process models which is typically enriched with additional information required for the execution of the process. In addition, the Eclipse-based plugin structure of JWT facilitates the implementation of the use cases and an integration with the Model Analysis Framework. Based on the metamodeling framework EMF, JWT employs the Graphical Editing Framework (GEF) to visualize the process models in its workflow editor component - which therefore also is the preferred method of presenting analysis results to the user.

Figure 10.1 and Figure 10.2 show excerpts of JWT's metamodel[3]. The elements in Figure 10.1 form the basis for modeling the control-flow while the classes in Figure 10.2 define additional annotations at ActivityNodes which encode information that is relevant for executing the process. The metamodel also provides means to organize processes and their associated elements in a UML-like package structure.

The root of the process graph is a Scope which contains ActivityNodes and ActivityEdges which form the basic control-flow. Scope is subclassed by Activity and StructuredActivityNode (abbreviated SAN). The latter type is a node which is part of a process and which itself contains an Activity. This subprocess is triggered every time the parent SAN is activated. The same is true for ActivityLinkNodes although, instead of embedding a subprocess in a parent process, they specify a reference to an existing Activity. Other relevant nodes types include Actions and ControlNodes. An Action represents the actual task or business function that has to be completed to continue with the execution of the process. ControlNodes are used to indicate the start and end points of a process (InitialNode and FinalNode), mutually exclusive paths (DecisionNode and MergeNode) and parallel paths (ForkNode and JoinNode).

Figure 10.2 is an excerpt of JWT's ReferenceableElement concepts, namely Roles, Applications and Data. These elements encode information required by the runtime environment - i.e. the process engine - to execute the modeled workflow. A Role, which is assigned to an Action, identifies the person who is responsible for carrying out the task. This could, for example, mean that a dialog form is displayed on the

[1] http://www.eclipse.org/jwt/

[2] [MR08] argues that most process designers use only about 10 of the core patterns provided by BPMN. These are also implemented by JWT (with the exception of Pool/Swimlane which can be simulated using JWT's Role concept). JWT allows to switch between different views on a process model which include - amongst others - representations as UML Activity Diagrams and BPMN processes. In addition, JWT models can be imported and exported from and to these formats through model transformations.

[3] Further information can be found at http://wiki.eclipse.org/JWT_Metamodel.

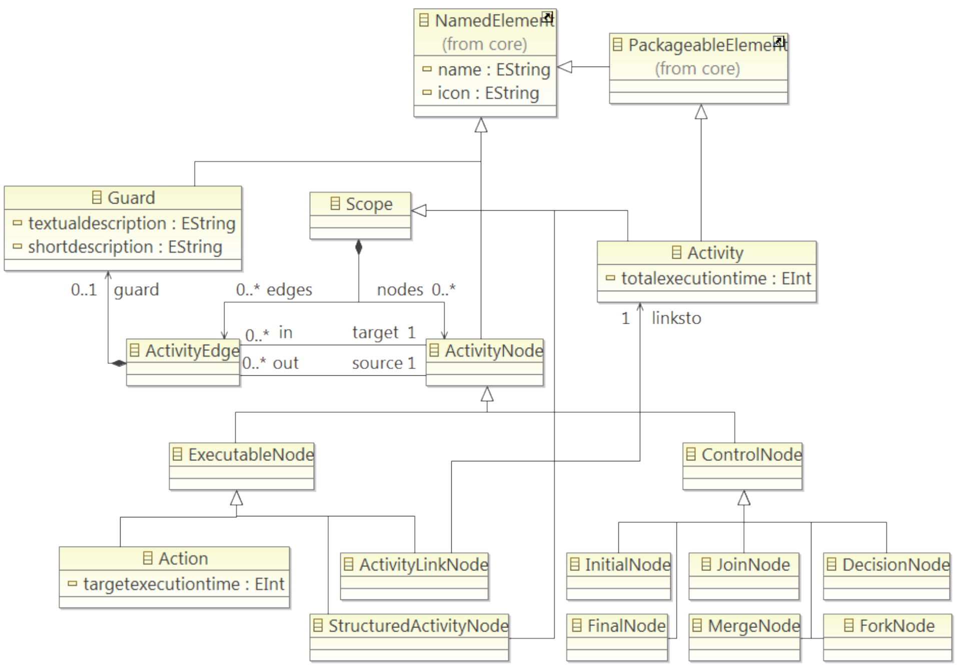


Figure 10.1.: JWT Metamodel: Elements of the core and processes packages.

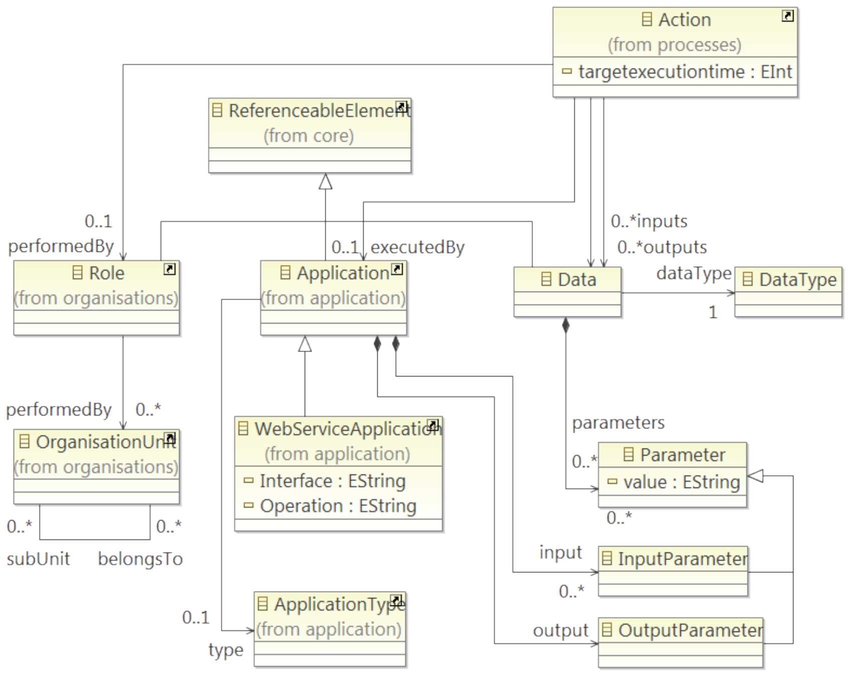


Figure 10.2.: JWT Metamodel: Referenceable elements Role, Application and Data

workstation of the respective person. An Application identifies the program (e.g. a webservice) which implements the Action while Data represents values that are either used or generated as indicated by the inputs and outputs relationships. All types of ReferenceableElements can be organized in a hierarchical Package structure (not shown in the metamodel excerpt).

Semantics of JWT processes with respect to Data-flow Analysis

Validating the static semantics of program code ensures that the statements form a valid definition with respect to the semantics of the programming language. In a similar way, well-formedness rules for business processes modeled in JWT must ensure that the processes can be executed by a workflow engine. This means that, in addition to providing all necessary data that the runtime environment requires to invoke the associated programs and services, the elements must adhere to a set of structural constraints:

- Each Activity must start with a single InitialNode and end with a FinalNode
- Each ActivityNode must possess exactly one incoming and one outgoing ActivityEdge with the following exceptions:
  - InitialNodes have no incoming edges
  - FinalNodes have no outgoing edges
  - DecisionNodes/ForkNodes may have one or more outgoing edges
  - MergeNodes/JoinNodes may have one or more incoming edges

The validation of these constraints is straightforward using e.g. OCL rules. However, there are also more complex restrictions that require the consideration of an element's context:

- All ActivityNodes must be reachable from the InitialNode and from each ActivityNode, the FinalNode must be reachable (cf. Section 10.1.2)[4].
- Parallel regions must be nested correctly (cf. Section 10.1.4).

Furthermore, certain best practices have evolved in the BPM domain:

- To improve maintenance and reuse, similar structures (clones) should be stored in separate subprocesses (cf. Section 10.1.5).
- Processes that are structurally and semantically sound may nevertheless violate established notions about how process models should be constructed. In this case, model metrics and guidelines can be used to assess these properties.

Applying DFA to JWT models requires special handling for the computation of data-flow properties for processes referenced via ActivityLinkNodes. In these cases, the same Activity may be referenced multiple times and can therefore be invoked in different contexts. Normally, this would lead to imprecise - or even incorrect - results as data-flow values would be aggregated for all contexts of the Activity. To enable a *context-sensitive analysis*, linked subprocesses therefore have to be converted to

[4]These rules have also been explored in the context of the running example in Section 6.1.3.

embedded processes (StructuredActivityNodes) by copying their contents. This handling is however restricted to process models in which the containment hierarchy of Scopes is non-cyclic.

To avoid the explicit computation of value domains for analyses which rely on the intersection operator to combine data-flow sets, we use the initialization marker *INIT* to denote the neutral element ($\top$) for which we implement specific handling in the data-flow rules. This increases the evaluation's performance by reducing memory and processing resources which would otherwise be required for the computation and the storage of these values. At the same time, this solution speeds up the execution of set operations in the case of large process models.

10.1.2. Use Case: Control-Flow Analysis

Since a BPMN-style business process inherently defines a control-flow, the application of many common DFA methods is straightforward. The benefit of the application of flow analysis is twofold: On the one hand, it is possible to validate the structure of a process to check whether it conforms to the basic requirements for control-flow graphs. On the other hand, the extracted data may contain information which helps domain experts in assessing certain properties of the modeled process. Alternatively, it can also serve as input for other algorithms as will be demonstrated in the following sections. First, we will take a look at how information about reachability/liveness, predecessor/successor sets and SCC properties can be derived from business processes. These use cases can therefore be regarded as a practical application and evaluation of the concepts described in Section 9.2.

Reachability and Liveness

A very basic requirement for business process models is that they must form a connected graph with a designated start node and a final state. In JWT, these concepts are implemented by the classes InitalNode and FinalNode. To ensure connectedness, each Action must be reachable from the InitialNode. If this is not the case, it would be impossible for it to be executed under any circumstances. Conversely, each Action must be live, i.e. there has to exist a path to the FinalNode. Otherwise, the execution of this element would result in a dead end.

The reachability/liveness properties are independent of the semantics of the execution path, i.e. one does not need to differentiate between alternative (DecisionNode / MergeNode) and parallel paths (ForkNode / JoinNode). Therefore, the relevant metamodel class for attribute annotation is ActivityNode as it covers all node types that are part of the model's control-flow. For this type, the reachability and the liveness properties can be implemented according to the definitions in Section 9.2.1.

Extending the analysis to include embedded Scopes requires to implement additional handling for StructuredActivityNodes. From the view point of their parent Activity, they act as normal Actions. However, the evaluation of the embedded Scope's contents depends on the results computed for the SAN itself: If the SAN cannot be reached, then all contained Actions are also not reachable. Therefore, the

InitialNode of an embedded Scope has to inherit the result computed at its parent. The successors of the SAN are only reachable if the FinalNode of the contained Scope can be reached. To accommodate for this, data-flow values must be propagated from the FinalNode of an embedded process to the successors of the parent StructuredActivityNode. This method has to be adapted accordingly for an implementation of the liveness analysis.

For the reachability analysis, the boolean result value for the fixed-point computation is initialized with false. The single exception is the InitialNode of the root Scope where it is set to true as this element represents the entry point of the overall process. For the liveness analysis, the same is true for the root Scope's FinalNode.

Algorithm 26 The attribution cfg_reachability

```
1: Attribution CFG_REACHABILITY
2:    attribute assignment isReachable : Boolean initWith false
3:    extend InitialNode with
4:       occurrenceOf isReachable calculateWith initialnode_isReachable
5:    extend ActivityNode with
6:       occurrenceOf isReachable calculateWith activitynode_isReachable
```

```
1: Rule INITIALNODE_ISREACHABLE(attrDef, context)
2:    if (context.container is SAN) then
3:       return context.container[isReachable]          ▷ inherit from SAN container
4:    return true                                     ▷ InitialNode reachable by default
```

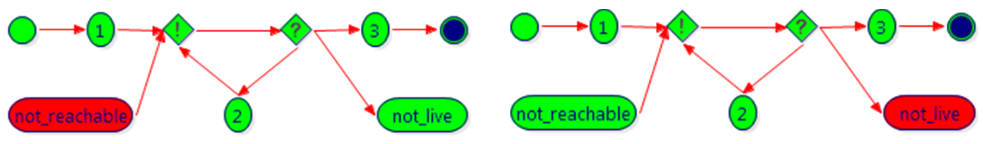
```
1: Rule ACTIVITYNODE_ISREACHABLE(attrDef, context)
2:    for all (sourceNode : context.in.source) do        ▷ iterate over source nodes
3:       sourceReachable ⇐ false
4:       if (sourceNode is SAN) then                    ▷ synthesize from subprocess
5:          finalNodesSAN ⇐ sourceNode.nodes→collect{FinalNode}
6:          sourceReachable ⇐ finalNodesSAN[isReachable]→contains{true}
7:       else
8:          sourceReachable ⇐ sourceNode[isReachable]      ▷ query source node
9:       if (sourceReachable) then
10:         return true                      ▷ true if at least one source is reachable
11:   return false                                              ▷ false otherwise
```

The attribution cfg_reachability[5], which implements this functionality, is shown in Algorithm 26. As mentioned, the isReachable attribute at InitialNodes inherits from its parent if it is part of a SAN while returning true for top-level processes[6]. Consequently results flow from a parent to the respective subprocesses in the process tree, mimicking the behavior of inh-type attributes in an attribute grammar. For Activ-

[5]cfg_liveness is structured similarly with the roles of InitialNode and FinalNode being reversed, i.e. information is requested from successors as opposed to predecessors.

[6]Note that this implementation does not require to explicitly initialize the root Scope's InitialNode with true.

ityNodes, all predecessors are considered. If the source node is a SAN, the function accesses the reachability property at the embedded FinalNodes, otherwise the value at the source node itself is used. In accordance with attribute grammar nomenclature, we refer to this principle as synthesis. As soon as at least one predecessor is identified as being reachable, true is returned for the current context.



(a) Reachability analysis (b) Liveness analysis

Figure 10.3.: Reachability/liveness analysis implemented for JWT.

The result of the implemented analysis is shown in Figure 10.3, demonstrating the validation of the reachability and liveness property for a JWT process.

Flow Sets

It can be assumed that, in addition to reachability and liveness, other control-flow properties such as predecessor/successor relationships and information about SCCs are also of interest to the user and may represent relevant inputs for subsequent analyses (cf. Section 9.2.2). For example, the approximation of an Action's potential predecessors indicates which steps may have taken place before the workflow reaches this point. Correspondingly, the set of guaranteed predecessors determines which Actions are guaranteed to have been executed at a specific point. This analysis is therefore able to assess whether certain preconditions are met on all possible execution paths.

This time - in contrast to the evaluation of the reachability/liveness properties - it is important to factor in the execution semantics of different path types. DecisionNodes impose an exclusive-or (xor) behavior on their outgoing edges as, each time the execution of a process reaches a split node, the execution will continue along exactly one of the available paths. At runtime, this decision is usually made based on the Guards annotated at the respective ActivityEdges. The evaluation of these statements is often context-sensitive as it may depend on the process' overall state (e.g. by accessing values of global variables). Parallel paths, which start with ForkNodes and are merged at JoinNodes, possess semantics somewhat similar to petrinets which use tokens to synchronize parallel executions [Aal98]. For JWT, this means that parallel paths have to be nested (overlapping paths are not allowed) and they must merge in a way that synchronization is always ensured. Otherwise, the process definition is invalid, potentially resulting in a lack of synchronization error[7].

For StructuredActivityNodes, the requirements are the same as in the previous use case: Nodes in embedded processes depend on the result which has been computed for their parent process and in turn propagate their results to the successors (or predecessors) of the SAN.

[7]This is further explored in Section 10.1.4.

The initialization value is set to *INIT* (representing $\top$) for both the minimal and maximal approximations instead of $\emptyset$ and the value domain respectively. This allows for a more compact definition of the data-flow rules as no initialization set must be computed. Additionally, this approach facilitates the integration of multiple analyses in a single attribution specification.

Algorithm 27 Data-flow rules of the attribution cfg_flowset

```
1: Rule ACTIVITYNODE_FLOW(attrDef, context, SAN_IN, SAN_OUT, FLOW_EDGE,
   FLOW_NODE, MAX)
2:    if (context is SAN_IN) and (context.container is SAN) then
3:       flowObjects ⇐ {context.container}                    ▷ use SAN container
4:    else
5:       flowObjects ⇐ context.FLOW_EDGE.FLOW_NODE         ▷ use direct neighbors
6:    for all (flowNode : flowObjects) do
7:       if (flowNode is SAN) then                     ▷ synthesize from subprocess
8:          flowSet ⇐ flowNode→collect{SAN_OUT}[attrDef.name]
9:       else                                              ▷ query direct neighbor
10:         flowSet ⇐ flowNode[attrDef.name]
11:      if (flowSet == INIT) then              ▷ preserve DFA initialization constant
12:         flowSets ⇐ flowSets + INIT
13:      else                      ▷ add neighbor and queried results to overall result
14:         flowSets ⇐ flowSets + (flowSet ∪ flowNode)
15:   if (flowSets→size == 1) and (flowSets[0] == INIT) then
16:      return INIT                            ▷ preserve DFA initialization constant
17:   if (MAX) or (context is JoinNode/ForkNode) then
18:      return ⋃(flowSets − INIT)                         ▷ maximal approximation
19:   return ⋂(flowSets − INIT))                           ▷ minimal approximation
```

```
1: Rule ACTIVITY_FLOW(attrDef, context, ACTIVITY_OUT, MAX)
2:    childFlowSets ⇐ context.nodes.ACTIVITY_OUT[attrDef.name]
3:    if (MAX) then
4:       return ⋃(childFlowSets)
5:    return ⋂(childFlowSets)
```

The data-flow rules in Algorithm 27 compute four result values for each ActivityNode and each Activity: allPredecessorsMax and allPredecessorsMin[8] represent the potential (maximal) and guaranteed (minimal) predecessors while allSuccessorsMax and allSuccessorsMin likewise define these sets for the successor analysis. It is obvious that the computation of each of these four results follows a very similar pattern. Therefore, instead of implementing four variants of the same rule, we employ a single method which can be parameterized.

The parameterization is shown in Table 10.1. The first two parameters for the generalized function activitynode_flow are required for the handling of SAN sub-

[8]allPredecessorsMin corresponds to the set of dominating nodes.

Target class	Data-flow rule	Parameterization
`allPredecessorsMin` : Set(ActivityNode)		
ActivityNode	activitynode_flow	InitialNode, FinalNode, in, source, false
Activity	activity_flow	InitialNode, false
`allPredecessorsMax` : Set(ActivityNode)		
ActivityNode	activitynode_flow	InitialNode, FinalNode, in, source, true
Activity	activity_flow	InitialNode, true
`allSuccessorsMin` : Set(ActivityNode)		
ActivityNode	activitynode_flow	FinalNode, InitialNode, out, target, false
Activity	activity_flow	FinalNode, false
`allSuccessorsMax` : Set(ActivityNode)		
ActivityNode	activitynode_flow	FinalNode, InitialNode, out, target, true
Activity	activity_flow	FinalNode, true

Table 10.1.: Parameterization for the flowset analysis.

structures. SAN_IN denotes the node type through which the *data-flow* enters the embedded process. For predecessor computations, this is the InitialNode and for successors the FinalNode type. Correspondingly, SAN_OUT denotes the class where results computed for the nested elements flow back to the parent process. The same principle applies to ACTIVITY_OUT in the signature of the activity_flow rule. FLOW_EDGE and FLOW_NODE are used to navigate to direct predecessors/successors using the references between ActivityNodes and ActivityEdges. If set to true, the boolean value MAX indicates that the maximal approximation shall be computed.

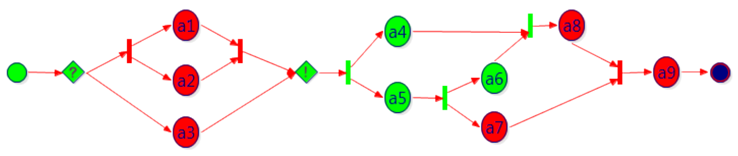
In order to calculate the result sets, the rule activitynode_flow first identifies the predecessors of of the context object. If the context is of the SAN_IN type (i.e. it receives results from its parent SAN), the container element is stored in flowObjects[9] [2 – 3]. Otherwise, line [5] stores the context's direct predecessors or successors. Subsequently, lines [6 – 14] request and process attribute values for the identified flowObjects. For SANs, the result is acquired from the respective SAN_OUT node [7 – 8]. By contrast, direct predecessor or successor nodes are queried directly [10]. If *INIT* is present in the requested results, the initialization constant is added to the overall result collection flowSets [12]. If this is not the case, line [14] adds the context node as well as the received results.

Computing the final value now only requires to unify (or intersect) the flowSets according to the parameterization. If all inputs of the context yield the *INIT* value, then it is also returned as result [15 – 16]. Otherwise, the rule distinguishes between potential and guaranteed results based on the value of the MAX variable [17 – 19]. One exception has to be made when computing min sets for parallel paths: Since all parallel paths must be traversed during the process' execution, line [17] applies the unification operator even if minimal sets are computed.

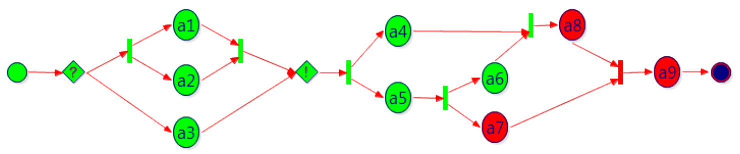
The rule activity_flow (which is optional) computes aggregated results for complete (sub)processes. For this purpose, it request the computation of ACTIVITY_OUT which corresponds to either the first or the last node of the respective Activity. This

[9]See rule initialnode_isReachable in Algorithm 26.

step will recursively trigger the evaluation of all depending nodes[10].



(a) allPredecessorsMin



(b) allPredecessorsMax

Figure 10.4.: Minimal and maximal predecessor sets for a8.

An example for the application of this analysis is shown in Figure 10.4. The two depictions of the process highlight the findings for the Action a8: Based on the results of the analysis, it can be guaranteed that - for any possible execution of this process - nodes { a4, a5, a6 } will have been visited before a8 is reached. Furthermore, taking into account all possible execution traces, { a1, a2, a3 } may additionally have been triggered beforehand[11]. By combining this information with the SCC detection, it can be determined whether the preceding Actions will have been executed exactly once, at least once or an arbitrary number of times.

Strongly Connected Components

In this use case, we will apply the SCC detection described in Section 9.2.3 to JWT models. In addition to indicating the presence of cyclic structures to the process designer, the computed results will also serve as input for subsequent analyses.

Just like in the flowset analysis, results are computed for both ActivityNodes and Activities. While not strictly necessary for the execution of the analysis, the aggregation of SCC results at Activities provides a quick overview of all nodes which are part of a cycle. However, results do not flow in or out of the child process (as is the case in the flowset analysis) since determining the global status of an Action is a trivial task: If a SAN is part of a cycle, all of the therein contained Actions automatically belong to the same (global) cycle. This information can therefore be retrieved by a simple traversal of the process tree. Computing the SCC status for each Activity's contents independently of its location with respect to parent processes therefore yields more fine-grained information. As the presence of SCCs is a property that depends only on the structural composition of a control-flow graph, it is not necessary to differentiate between alternative and parallel paths.

[10]This requires that all nodes are connected.
[11]In this case, either a1 **and** a2 or only a3 will be executed before a8.

Algorithm 28 The attribution cfg_scc

```
1: Attribution CFG_SCC
2:    attribute assignment sccID : Integer initWith 0
3:    attribute assignment sccObjects : Set(ActivityNode) initWith $\emptyset$
4:    extend ActivityNode with
5:       occurrenceOf sccID calculateWith activitynode_sccID
6:       occurrenceOf sccObject calculateWith activitynode_sccObjects
7:    extend Activity with
8:       occurrenceOf sccID calculateWith activity_sccID
9:       occurrenceOf sccObject calculateWith activity_sccObjects
```

```
1: Rule ACTIVITYNODE_SCCID(attrDef, context)
2:    if (context is SAN) then
3:       activity_sccID(attrDec, context)                    ▷ trigger subprocess evaluation
4:    if (context[allPredecessorsMax]→contains{context}) then
5:       return contextAllPredecessors→hashValue             ▷ generate SCC hash
6:    return 0                                               ▷ node not in SCC
```

```
1: Rule ACTIVITY_SCCID(attrDef, context)
2:    return $\bigcup$(context.nodes[sccID])                  ▷ trigger evaluation for nodes
```

```
1: Rule ACTIVITYNODE_SCCOBJECTS(attrDef, context)
2:    if (context is SAN) then
3:       activity_sccObjects(attrDec, context)                ▷ trigger evaluation for nodes
4:    if (context[sccID] != 0) then                           ▷ only if node belongs to SCC
5:       for all predNode : context.in.source) do             ▷ process predecessors
6:          if (predNode[sccID] == context[sccID]) then       ▷ same SCC hash
7:             sccObjects ⇐ sccObjects ∪ predNode[sccObjects] ∪ context
8:    return sccObjects
```

```
1: Rule ACTIVITY_SCCOBJECTS(attrDef, context)
2:    return $\bigcup$(context.nodes[sccObjects])
```

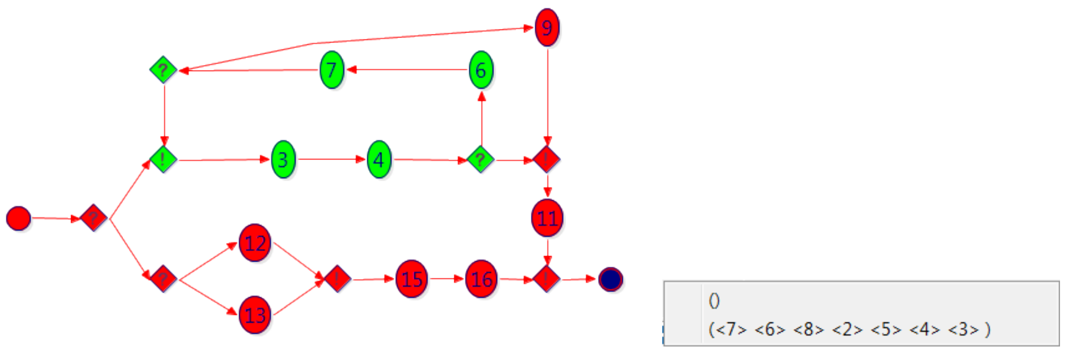
The attribution which implements the SCC detection is listed in Algorithm 28. It defines the two attributes sccID and sccObjects which are responsible for computing the SCC state at the classes Activity and ActivityNode.

To calculate the cycle identifier, the rule activitynode_sccID requests the set of potential predecessors allPredecessorsMax of the context node and checks if the context is part of that set $[4-5]$. If a node is found to be its own predecessor, it is contained in a cycle and the hash value of the set of all predecessors (which is the same for every node of the respective SCC) is used as an identifier[12]. Lines $[2-3]$ of the rule trigger the evaluation of embedded subprocesses at SANs. This functionality has

[12]Alternatively, the local allPredecessorsMax value could be compared against the unification of allPredecessorsMax at the preceding nodes in the control-flow as both sets have to contain the same elements for nodes which are part of the same SCC.

been added for convenience reasons to automatically evaluate the process subtree of any given Activity. For each Activity[13], the rule activity_sccID creates the union of all SCC identifiers[14], thereby triggering the evaluation of the contained elements.

To compute sccObjects, the rule activitynode_sccObjects iterates over all direct predecessor nodes of the context object (lines [5 – 7]) if the local node is part of a cycle [4]. If the preceding ActivityNode is part of the same SCC [6], its sccObjects result is requested and extended by the current context [7]. This process iteratively builds a set of all SCC objects over the course of the fixed-point evaluation of the attribute. Due to the conceptual similarities, activity_sccObjects is structured in the same way as activity_sccID.



()
(<7> <6> <8> <2> <5> <4> <3>)

(a) sccID (b) sccObjects

Figure 10.5.: Identified SCC in a JWT process.

The application of this analysis is demonstrated in Figure 10.5. The identified SCC elements have been highlighted in green in the depiction of the process in Figure 10.5(a). The screenshot in Figure 10.5(b) shows the two unique sccObject sets which have been extracted from the analysis results. The empty set ∅ is available at each node which does not belong to a cycle while the set containing the ActivityNodes ② to ⑧ is returned for all nodes of the identified SCC.

SCC Ports

The following analysis computes another set of relevant SCCs properties: Ports are nodes which connect a cyclic structure to the surrounding environment.

If a cycle is part of a process graph in which every element is reachable and live and the overall number of process nodes is larger than the amount of nodes that are part of the SCC, then the cycle contains nodes that are connected through edges with nodes outside the SCC. Following the naming conventions of [Got+09], these are termed ports as they provide a "bridge" between the SCC and the remainder of the process. We will refer to the incoming and outgoing edges of these ports as in

[13]An attribution of Scope (which is a superclass of Activity and SAN) would not be possible in this case as this would lead to an incoherent attribute inheritance structure: Due to multiple inheritance, SAN objects would be associated with two occurrences of the same attribute.

[14]Since the computation relies solely on the precomputed predecessors, there would be no benefit in adhering to a specific order, e.g. starting at InitialNodes.

edges and out edges of the SCC respectively. Nodes and edges that belong to a cycle but do not fulfill these properties are named inner edges and inner nodes.

The JWT modeling language requires the presence of dedicated InitialNodes and FinalNodes. Additionally, multiple incoming and outgoing connections are only allowed for gateways. It is also not possible to enter or leave a cycle at parallel paths. As a consequence, the well-formedness rules only allow for DecisionNodes and MergeNodes to be connection points of cycles. For this reason, any SCC in a JWT model possesses at least one input and one output port of these types. For each port one or more in and out edges may exist.

As ports are properties of a SCC and the following analysis is based on the cfg_scc attribution, results are again computed on a Scope/Activity level rather than inheriting the SCC status to embedded processes in order to achieve more significant results. The evaluation starts out with empty sets for the attribute values. The attribution that identifies the in/out edges is shown in Algorithm 29 while the ports are computed by Algorithm 30. As with cfg_flowset, parameterized rules are used - in this case to combine the calculation for the in and out cases. The parameterization is listed in Table 10.2.

Target class	**Data-flow rule**	**Parameterization**
sccInEdges : Set(ActivityEdge)		
ActivityEdge	activityedge_sccEdges	in, source, target
ActivityNode	activitynode_sccEdges	in
Activity	activity_sccEdges	
sccOutEdges : Set(ActivityEdge)		
ActivityEdge	activityedge_sccEdges	out, target, source
ActivityNode	activitynode_sccEdges	out
Activity	activity_sccEdges	
sccInPorts : Set(ActivityNode)		
ActivityEdge	activityedge_sccPorts	source, target
ActivityNode	activitynode_sccPorts	in, sccInEdges
Activity	activity_sccPorts	
sccOutPorts : Set(ActivityNode)		
ActivityEdge	activityedge_sccPorts	target, source
ActivityNode	activitynode_sccPorts	out, sccOutEdges
Activity	activity_sccPorts	

Table 10.2.: Parameterization for the port analysis.

The reference types FLOW_SOURCE and FLOW_TARGET are used to navigate to the source and the target node of an edge depending on the intended flow direction. To compute edges and ports through which a cycle can be entered, these parameters must be set to source and target respectively and vice versa for outgoing elements. The parameter FLOW correspondingly denotes the in or out reference of ActivityNodes.

Algorithm 29 Data-flow rules of the attribution cfg_ports

```
1: Rule ACTIVITYEDGE_SCCEDGES(attrDef, context, FLOW, FLOW_SOURCE,
   FLOW_TARGET)
2:     sourceSCCID ⇐ context.FLOW_SOURCE[sccID]      ▷ SCC hash from (flow) source
3:     targetSCCID ⇐ context.FLOW_TARGET[sccID]      ▷ SCC hash from (flow) target
4:     if (sourceSCCID == 0) or (sourceSCCID != targetSCCID) then
5:         return ∅        ▷ edge not in SCC or source/target belong to different SCCs
6:     for all (flowEdge : context.FLOW_SOURCE.FLOW) do
7:         if (flowEdge[sccID] == sourceSCCID) then      ▷ source edge in same SCC
8:             sccEdges ⇐ sccEdges ∪ flowEdge[attrDef.name]      ▷ forward results
9:         else
10:            sccEdges ⇐ sccEdges ∪ flowEdge                  ▷ add in/out edge to result
11:    return sccEdges
```

```
1: Rule ACTIVITYNODE_SCCEDGES(attrDef, context, FLOW)
2:     if (context is SAN) then                      ▷ trigger subprocess evaluation
3:         activity_sccEdges(attrDef, context)
4:     return ⋃(context.FLOW[attrDef.name]      ▷ combine results in flow direction
```

```
1: Rule ACTIVITY_SCCEDGES(attrDef, context)
2:     return ⋃(context.nodes[attrDef.name])
```

The rule activityedge_sccEdges first acquires the sccID of the source and target node of the current edge $[2-3]$. Lines $[4-5]$ ensure that the empty set is returned if the edge is not part of a SCC or the connected nodes belong to two different cycles. As a consequence, the following computation is only carried out for inner edges. The loop in lines $[6-10]$ then iterates over all incoming edges at the source node (alternatively all outgoing edges at the target node) to determine which edges enter (or leave) the cycle. For inner edges, lines $[7-8]$ unify the already computed results, thereby propagating them throughout the SCC structure. Entering and leaving edges are added to the result set [10]. The rule activitynode_sccEdges triggers the computation of SAN subprocesses in lines $[2-3]$ and returns the union of the sets computed for its incoming (or outgoing) edges.

Algorithm 30 Data-flow rules of the attribution cfg_ports (continued)

```
1: Rule ACTIVITYEDGE_SCCPORTS(attrDef, context, FLOW_SOURCE, FLOW_TARGET)
2:     sourceSCCID ⇐ context.FLOW_SOURCE[sccID]          ▷ SCC hash from (flow) source
3:     targetSCCID ⇐ context.FLOW_TARGET[sccID]          ▷ SCC hash from (flow) target
4:     if (sourceSCCID == 0) or (sourceSCCID != targetSCCID) then
5:         return ∅          ▷ edge not in SCC or source/target belong to different SCCs
6:     return context.FLOW_SOURCE[attrDef.name]          ▷ inside SCC: forward results
```

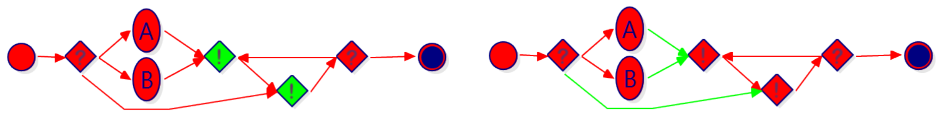
```
1: Rule ACTIVITYNODE_SCCPORTS(attrDef, context, FLOW_SOURCE, EDGES)
2:     if (context is SAN) then                          ▷ trigger subprocess evaluation
3:         activity_sccPorts(attrDef, context)
4:     return ⋃(context[EDGES]).FLOW_SOURCE              ▷ compute result from in/out edges
```

```
1: Rule ACTIVITY_SCCPORTS(attrDef, context)
2:     return ⋃(context.nodes[attrDef.name])
```

The ports can now be derived from the edge results. The rule activitynode_sccPorts requests the edge results for the current context, flattens them into a single set and returns their respective source or target nodes. activityedge_sccPorts now only has to relay the result computed at its source or target if the context is an inner edge of a SCC.

Obviously, this approach could also be implemented the other way around with the in/out edge rules deriving their results from the previously computed ports. Additionally, it would also be possible to extend this attribution to compute the inner edges of SCCs and make them available at every node.

One important aspect to consider in this use case is that the computation of cfg_ports requires that cfg_scc has already been executed. Otherwise, a preliminary result for sccID might lead to an edge being falsely identified as an in/out edge and included in the results that are then distributed amongst the SCC nodes. In this case, the wrongly identified edge will stay in the result set even after the final value for sccID becomes available.



(a) sccInPorts (b) sccInEdges

Figure 10.6.: Identified input ports and in edges of a SCC in a JWT process.

An example of this use case is depicted in Figure 10.6. The SCC in this process has two input ports, one of which has two and the other a single incoming edge, resulting in three in edges overall. By forwarding these attribute values, the results are available at each node and each edge belonging to the respective SCC and can be easily accessed if required for subsequent analyses.

10.1.3. Use Case: Definition/Usage of Data Objects

A major advantage of business process models is their executability by means of a workflow engine. Objects assigned to JWT's business actions can trigger the execution of associated applications, supply them with required input data and manage the generated output which can then again be passed on as parameters to other programs. For this analysis, we will focus on latter type of objects, the Data elements, which are interpreted as parameters by the workflow engine.

From the viewpoint of static analysis, parameters assigned to different nodes in the control-flow form input/output relationships that can be subjected to an approximation concerning their availability at subsequent steps. The basic premise of this concept is that once a specific data object has been generated, it will be available at other actions (being stored in memory or a database) until it is consumed. This otherwise straightforward transfer of information is complicated by the use of gateways which add alternative paths and cycles to the control-flow in which case the availability of data elements during runtime must be approximated.

These properties make this analysis an exemplary use case for DFA. In fact, it bears strong similarities to the reaching definitions [AC76] analysis which is one of the oldest and most commonly cited applications of data-flow analysis. However, unlike reaching definitions, we will not analyze which exact definition of a data object reaches subsequent process steps as this information would not be very valuable in this context[15]. Instead, we will focus on the general availability status which is characterized by the two states of data guaranteed to be available on any execution path and data that might be available on a subset of paths. The goal here is to support process designers by identifying missing inputs and to draw attention to information that might be generated but will not be used subsequently.

[15]In contrast to program code, the semantics of external applications is not encoded in the process itself. Because, in this case, the analysis aims to validate rather than optimize, little is to be gained by determining the availability of results based on their specific point of origin.

Algorithm 31 Data-flow rules for the attributions availableData and unusedData

```
 1: Rule ACTIVITYNODE_AVAILABLEUNUSEDDATA(attrDef, context, MAX)
 2:     if (context is InitialNode) and (context.container is SAN) then
 3:         relevantNodes ⇐ context.container.in.source      ▷ inherit from container
 4:     else if (context is SAN) then
 5:         relevantNodes ⇐ context.nodes→collect{FinalNode}         ▷ synthesize
 6:     else
 7:         relevantNodes ⇐ context.in.source              ▷ query predecessor nodes
 8:     inDataSets ⇐ relevantNodes[attrDef.name]
 9:     if (inDataSets→size == 1) and (inDataSets[0] == INIT) then
10:         return INIT                              ▷ preserve initialization value
11:     if (MAX) or (context is JoinNode) then
12:         result ⇐ $\bigcup$(inDataSets)                 ▷ union for max/parallel paths
13:     else
14:         result ⇐ $\bigcap$(inDataSets)              ▷ intersection for everything else
15:     return result − INIT
```

```
 1: Rule ACTION_AVAILABLEUNUSEDDATA(attrDef, context, UNUSED, MAX)
 2:     result ⇐ activitynode_availableUnusedData(attrDef, context, MAX)
 3:     result ⇐ result ∪ context.outputs                 ▷ add produced outputs
 4:     if (UNUSED) then
 5:         result ⇐ result − context.inputs             ▷ remove consumed outputs
 6:     return result
```

The specifications for the computation of the attributes availableData and unusedData for both the minimal and maximal approximations are very similar: The result type is a set of Data objects and occurrences of the respective attribute are calculated at elements of the types Activity, ActivityEdge and ActivityNode (which is overwritten at instances of Action). The initialization marker *INIT* and the empty set are used as initial values for the computation of guaranteed/potential availability information respectively. The respective attribute type (available/unused, min/max) is configured through a parameterization (cf. Table 10.3) of the rules activitynode_availableUnusedData and action_availableUnusedData (cf. Algorithm 31).

As with other use cases in this chapter, activitynode_availableUnusedData takes its inputs from the direct neighbors of the context object (in this case the incoming edges), passes results on to subprocesses and propagates them back to the parent process [2 − 8]. In lines [9 − 10], the *INIT* marker (representing the $\top$ value) is preserved if it is received on all paths. Since the semantics of parallel paths require that all incoming paths arrive at the synchronization point before continuing, the sets containing the available data are merged at JoinNodes. The same is true if the desired output should yield the maximal availability [11 − 12]. Otherwise, [14] intersects the sets to eliminate information which is not received on all incoming paths. Finally, it is ensured that the *INIT* marker is removed from the result.

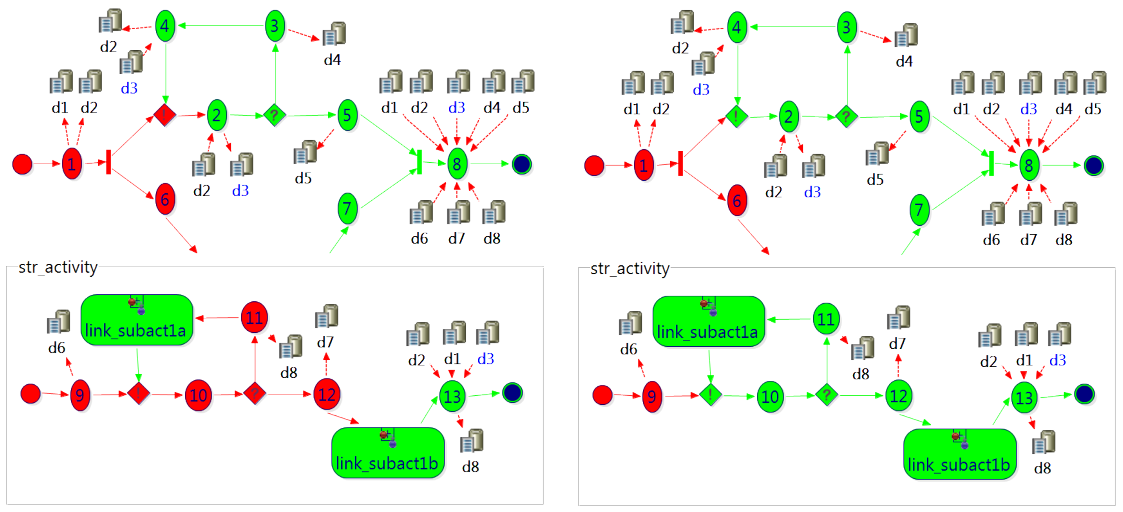
The computation logic for Action elements extends the behavior employed for the

Target class	Data-flow rule	Parameterization
`availableDataMin` : Set(Data)		
ActivityNode	activitynode_availableUnusedData	false
Action	action_availableUnusedData	false, false
`availableDataMax` : Set(Data)		
ActivityNode	activitynode_availableUnusedData	true
Action	action_availableUnusedData	false, true
`unusedDataMin` : Set(Data)		
ActivityNode	activitynode_availableUnusedData	false
Action	action_availableUnusedData	true, false
`unusedDataMax` : Set(Data)		
ActivityNode	activitynode_availableUnusedData	true
Action	action_availableUnusedData	true, true

Table 10.3.: Parameterization for the available and unused data analysis.

evaluation of ActivityNodes. Therefore, action_availableUnusedData first invokes the former rule (line [1]) before adding the data objects which are generated by the local Action to the result [2]. If the set of unused Data should be computed, the elements which are consumed by the local Action must be subtracted from the result set [4 – 5].

Available Data



(a) Minimal approximation (availableDataMin) (b) Maximal approximation (availableDataMax)

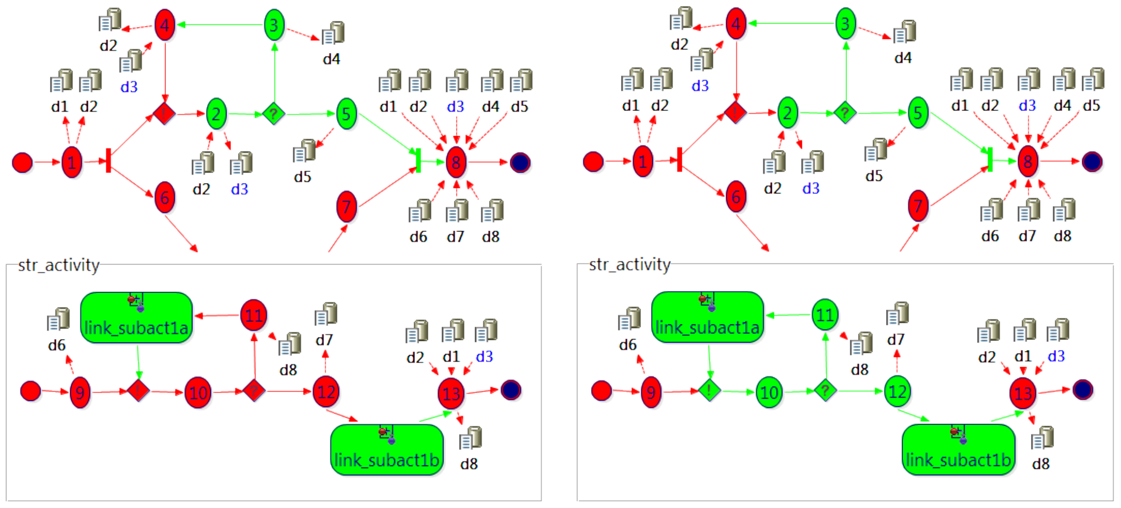
Figure 10.7.: Approximation of the propagation of Data d3.

The application of the availability analysis is demonstrated in Figure 10.7: The displayed process model defines a set of data elements {d1, . . . d8} which are produced and consumed by the business actions. In this example, the color denotes the availability status of the Data object $\boxed{\text{d3}}$ which is produced by the Action ② and inside the subprocesses of the StructuredActivityNodes *link_ subact1a* and *link_ subact1b* and is read at ④, ⑧ and ⑬.

The result for the minimal availability of $\boxed{\text{d3}}$ is visualized in Figure 10.7(a). Consequently, the highlighted parts denote elements at which this object will be available in any execution of the process (*guaranteed availability*). This concept can be explained using the example of the MergeNode which is the direct predecessor of Action ②: While $\boxed{\text{d3}}$ is available at all successors after having been generated at ②, it is not propagated to the cycle's entry point since there exists an execution path (the first time the execution arrives at this node) where it is not (yet) available. If the Data element is produced inside cycles or alternative paths, it does not leave these regions as can be seen at *link_ subact1a*.

In most cases, the minimal approximation represents a suitable, conservative solution as the computed availability status of data objects is guaranteed to hold for all possible executions of the process. However, it is also conceivable that one might be interested in the maximal availability, especially when combined with the analysis of unused data (see below). Figure 10.7(b) illustrates the results if the maximal approximation is used, thus yielding the points where $\boxed{\text{d3}}$ might be available considering all possible executions of the process (*potential availability*). In other words, for each element at which $\boxed{\text{d3}}$ is determined to be available, there exists at least one execution path where this is actually the case.

Unused Data



(a) Minimal approximation (unusedDataMin) (b) Maximal approximation (unusedDataMax)

Figure 10.8.: Approximation of the unused paths of Data object $\boxed{\text{d3}}$.

The results for the unusedData attributes - again with the visualization focusing on $\boxed{\text{d3}}$ - can be seen in Figure 10.8. Because the rule action_availableUnusedData removes data as soon as it is read by an Action, the highlighted nodes represent the paths alongside which the data objects are transported before being accessed for the first time.

Again, the analysis may either yield the minimal results that are guaranteed to hold for any execution of the process (cf. Figure 10.8(a)) or compute an approximation for all possible execution paths (cf. Figure 10.8(b)). This choice affects the propagation of information about unused objects on alternative or optional paths. In the example, we can see how this principle applies to the nodes inside the embedded process *str_activity*: For unusedDataMax, ⓓ3 is reported as unused for all subsequent nodes until it is read by ⑬. One could argue that for use cases such as determining how many resources must be allocated to buffer data until it is read, the maximal approximation represents the conservative approximation.

Missing Data

For a process designer, it is useful to know whether a data object will be available at business actions where it is required as input. Because of alternative and cyclic paths, this information must be approximated, since the analysis has to consider all execution paths while a human expert might be able to assert that certain problematic paths will never be reached in practice due to detailed knowledge about the modeled business domain.

Algorithm 32 Data-flow rules for the attribution missingData

```
1: Rule ACTION_MISSINGDATA(attrDef, context, MINMAX)
2:     if (context[MINMAX] != INIT) then
3:         return context.inputs − context[MINMAX]
4:     return INIT
```

```
1: Rule SCOPE_MISSINGDATA(attrDef, context)
2:     actionsAndSANs ⇐ context.nodes→collect{Action,SAN}
3:     return ⋃(actionsAndSANs[attrDef.name])
```

Target class	Data-flow rule	Parameterization
missingDataMin : Set(Data)		
Action	action_missingData	availableDataMin
Scope	scope_missingData	
missingDataMax : Set(Data)		
Action	action_missingData	availableDataMax
Scope	scope_missingData	

Table 10.4.: Parameterization for the missing data analysis.

The data-flow rules in Algorithm 32 compute the set of missing data elements based on the results of the availability analysis. The parameterization is listed in Table 10.4. The computation is carried out locally at Actions by requesting the values of availableDataMin or availableDataMax at the local object and subtracting the available data from the inputs. Any input data not in the set of availableDataMin/availableDataMax is thereby marked as missing. Scopes simply collect all computation results for easy access in post-analysis processing.

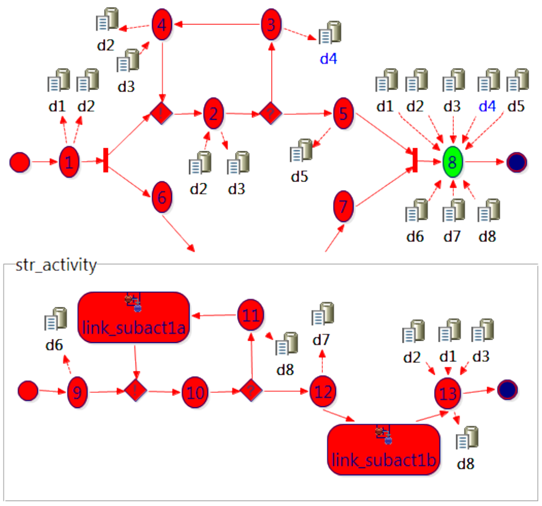


Figure 10.9.: Missing Data d4 based on the minimal approximation.

Figure 10.9 shows the results of missingDataMin for the example process. In this case, a (potentially) erroneous access to $\boxed{\text{d4}}$ is reported for the Action ⑧. Since the results are based on the minimal approximation, this does not imply that the error occurs on every process execution. Instead, the result indicates that there exists at least one erroneous path. In the example process, no problem occurs if the cycle {②,③,④} is traversed at least once before reaching node ⑧. As has been mentioned, based on the knowledge about the modeled business process, a process designer might either decide that this is in fact a problem within the process definition or that an assumption can be made that the cycle will always be executed at least once and therefore dismiss the problem report.

The application of missingDataMax to this process does not yield any errors. Because this attribute computes the maximal approximation, all possible propagation paths are considered in the computation of the missing data. Any reported problem would therefore occur on every possible execution of the process. As a consequence, we can conclude that the example process has at least one execution path which will not result in errors due to missing input data.

Conclusions and Outlook

There are many conceivable improvements that can be implemented to the extend the capabilities of the presented analyses and thereby the value of the generated information:

- As with the use cases described in the previous sections, it would be possible to reverse the direction of the data flow, turning the forward into a backward analysis. Currently, the result indicates definition/usage relationships between the occurrences of data objects at business actions. A backward analysis would yield information about usage/definition relations between data objects. This would improve the detection of unused data by factoring in the last (instead of the first) location where a parameter is accessed.

- Further improvements can be made by including guards at transitions: For example, if a data object is of the type integer and a guard restricts the execution of a specific path to cases where the value is constrained to a predefined range, then we can deduce that all data generated on this path will only be available if this condition is met, thereby producing a more detailed report.

- The analysis can also be extended to record which Roles perform the read/write operations to track which person was the last to access or modify a specific data object for each step of the process.

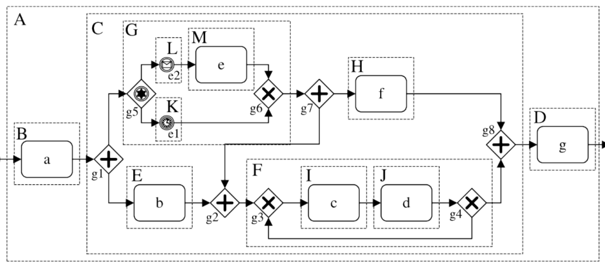
- Some resources might not support parallel write access or must be protected as long as one of multiple concurrent paths might alter their values. In this case, different instances of the same object could be tracked inside parallel regions to determine potential conflicts.

- The unused analysis removes data from the result set on its first access. This behavior can be altered to implement different read/consume semantics for data accesses to reflect the execution semantics of the respective workflow engine or the backend storage facilities.

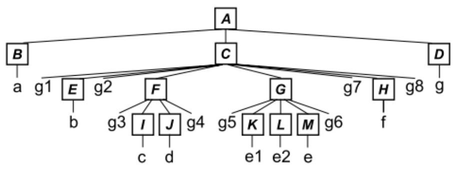
10.1.4. Use Case: Decomposition, Validation and Transformation

A common requirement when dealing with control-flow graphs is the decomposition of a graph into smaller fragments or components. The act of breaking down a control-flow into its constituents is motivated by the fact that smaller subgraphs are often easier to process. Structural decomposition may also be necessary for the application of certain algorithms. An example for this can be found in [Lau10] where structural information is used to improve the performance of an automated process planning algorithm: If the underlying model changes, the process is first decomposed to identify the smallest component encompassing all modified elements for which the planning process can then be repeated.

The Single Entry Single Exit (SESE) mechanism provides a convenient way for dividing a control-flow into (sub)components: SESE components (also called regions or fragments) represent a subdivision of a control-flow graph into hierarchically structured, non-overlapping subgraphs with exactly one entry and one exit node. An example of a SESE decomposition and the resulting structure (called a process structure tree in the domain of business process modelling) can be seen in Figure 10.10. In this use case, we will develop a unified approach for the structural processing of control-flow graphs based on data-flow analysis. This encompasses the hierarchical decomposition of business processes resulting in a tree of SESE



(a) SESE regions



(b) Program structure tree

Figure 10.10.: Single Entry Single Exit decomposition of a business process [Gar08].

components, the structural validation of processes and their transformation into a block-oriented representation.

Token Analysis

To perform the decomposition, the authors of [Got+09] propose an algorithm which assigns labels - named tokens - to the transitions of a flow graph. The tokens are created at split nodes (alternatively called gateways), i.e. nodes with multiple outgoing edges, and are propagated in the direction of the control-flow. Matching sets of tokens are subsequently merged if they fulfill certain conditions.

In detail, the algorithm works as follows: First, tokens are created and propagated. The group of tokens (hereafter referred to as tokenset) created at a split node n_{split} contains a token $t_{(n_{split},index)}$ for each outgoing edge where *index* is a unique identifier in the context of this tokenset. Each token therefore carries with it information about the node at which it has been created. At nodes with exactly one predecessor, the incoming tokenset is simply forwarded while nodes that merge multiple paths also lead to the unification of the respective sets. In the second step of the algorithm, tokens are removed from an edge's tokenset if all tokens created at a split node are available at this edge. This process (referred to as converging tokens) results in tokens canceling each other out if all tokens originating from a specific split node are contained in the same tokenset. This way, once all paths originating from a gateway converge, the token labeling reverts to the state is has been before the split. This principle also holds true for nested control structures.

A special handling must be implemented for cyclic paths: Because of the forwarding of tokensets, cycles as a whole act as if they were nodes. Depending on the amount of edges leaving a cycle, they behave as either sequential nodes or gateways. The structure inside a cycle thus cannot be analyzed as tokens created at cyclic gateway nodes would propagate throughout the whole cycle, converging at each node. Instead, the analysis has to be executed recursively: While the remainder of the graph still contains a SCC, its ports have to be removed and the analysis must be repeated for the inner nodes and edges.

When the final token labelings have been established, the SESE components can be detected by identifying their entry and exit points. These are marked by edges which carry the same token labelings. The exception are sequences of nodes which are characterized by multiple nodes with one incoming and one outgoing edge carrying identical tokensets. This algorithm ensures that the SESE components are always fully contained inside each other. Consequently, this analysis results in a structure tree with SESE components as nodes and the control-flow elements as its leaves.

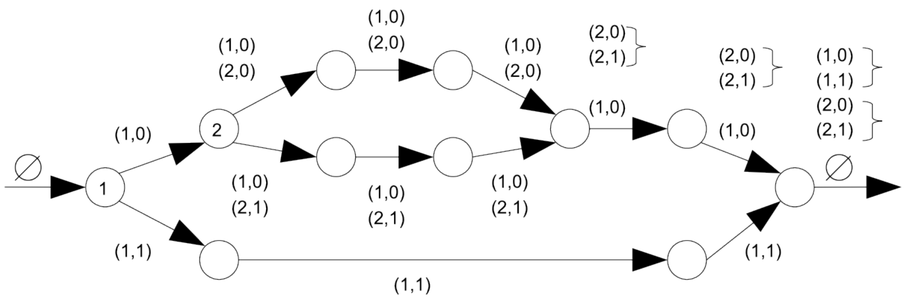


Figure 10.11.: SESE decomposition of control-flow graphs using token-flow analysis (initial tokensets and token convergence) [Got+09].

The application of this algorithm is demonstrated in the example shown in Figure 10.11. Tokens $t_{(1,0)}$ and $t_{(1,1)}$ are generated at the outgoing edges of split node ① and propagated along the flow edges. The same principle applies at node ② resulting in the tokens $t_{(2,0)}$ and $t_{(2,1)}$. In the second step, the tokensets $\{(1,0),(1,1)\}$ and $\{(2,0),(2,1)\}$ converge (indicated by curly brackets), resulting in the final token labeling.

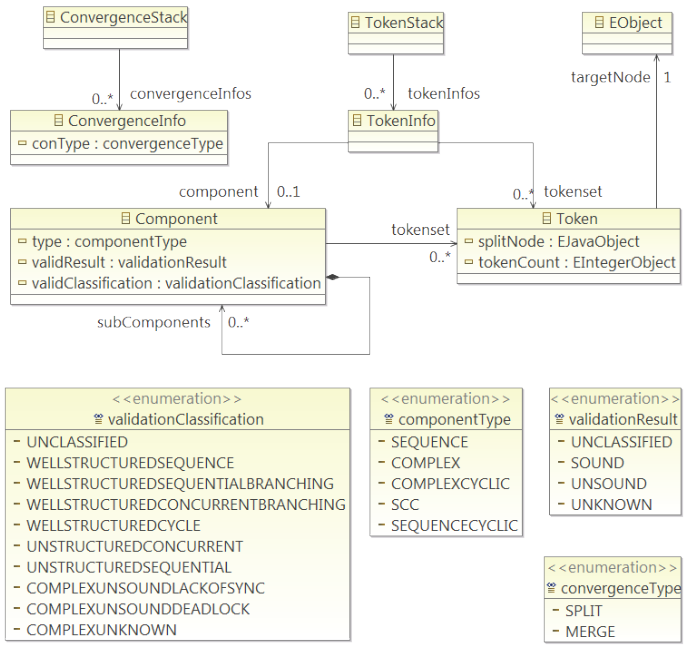


Figure 10.12.: The metamodel used by the tokenflow attributions.

The authors of [Got+09] have provided an imperative implementation of the algorithm which uses timestamps to manage the visitation order and contracts identified

SCCs into single nodes. Since the calculation of the tokens relies heavily on information propagation, this algorithm can be easily realized as a declarative specification based on the data-flow analysis approach.

Because tokens and the subsequently derived components represent complex datatypes, they can be described by a dedicated metamodel which is shown in Figure 10.12. In addition to defining Token and Component classes, this metamodel also implements additional features which are required by other analyses. For example, Components possess fields for storing their respective structural type (componentType) and validation results (validationResult, validationClassification) alongside the token labeling which identifies the component and its children. TokenStack and TokenInfo can be used to perform a DFA-based component detection while ConvergenceStack and ConvergenceInfo are used to implement the partial convergence feature of the tokenflow approach. By extending the generated model code, these datatypes can implement custom semantics for specific operations such as hash code generation or instance equality checks.

The attribution tf_token, listed in Algorithm 33, provides a DFA-based implementation of the token analysis[16]. It declares two attributes, tokensInitial and tokensFinal, which are responsible for token propagation and convergence respectively. Both attributes are bound to the class ActivityEdge. Furthermore, the attribution relies on sccID from cfg_scc (and by extension allPredecessorsMax from cfg_flowset) and sccOutEdges from cfg_ports.

The data-flow rule activityedge_tokensInitial computes the set of initial tokens for each element of the type ActivityEdge. First, in lines $[5-6]$, it is determined whether the current edge is located inside a cycle. This is the case if sccID at the source and the target node of the edge have the same value and if this value is not zero. Line [7] then combines all tokensets which reach the source node on its incoming edges into a single tokenset. If the source node is not a split node, this value represents the result. Otherwise, a new token is created for the source gateway and the current edge. Depending on whether the source node of the edge is part of a cycle, this happens in lines $[11 - 12]$ or $[14 - 18]$. Finally, line [19] creates the token object as an instance of the Token class from the metamodel in Figure 10.12 and adds it to the merged tokenset.

[16]Appendix C.1.1 contains an alternative implementation of the data-flow rules using imperative OCL.

Algorithm 33 The attribution tf_token

```
Attribution TF_TOKEN
    attribute assignment tokensInitial : Set(Token)
        initWith ∅
    attribute assignment tokensFinal : Set(Token)
        initWith ∅
    extend ActivityEdge with
        occurrenceOf tokensInitial
            calculateWith activityedge_tokensInitial
        occurrenceOf tokensFinal
            calculateWith activityedge_tokensFinal
```

```
Rule ACTIVITYEDGE_TOKENSINITIAL(attrDef, context)
    sourceNode ⇐ context.source
    targetNode ⇐ context.target
    contextSCCID ⇐ 0                                          ▷ edge not inside SCC
    if (sourceNode[sccID] == targetNode[sccID]) then
        contextSCCID ⇐ sourceNode[sccID]                       ▷ edge inside SCC
    tokens ⇐ ⋃(sourceNode.in[tokensInitial]) − INIT           ▷ combine tokensets
    if (contextSCCID == 0) and (sourceNode.out→size > 1) then
        sourceNodeSCCID ⇐ sourceNode[sccID]
        if (sourceNodeSCCID == 0) then                         ▷ create split gateway token
            splitNode ⇐ sourceNode
            tokenCount ⇐ sourceNode.out→size
        else                                                   ▷ create SCC token
            sccOutEdges ⇐ sourceNode[sccOutEdges]
            if (sccOutEdges→size == 1) then                    ▷ SCC has only one out edge
                return tokens
            splitNode ⇐ sourceNodeSCCID
            outgoingCount ⇐ sccOutEdges→size
        tokens ⇐ tokens ∪ new Token(splitNode, targetNode, tokenCount)
    return tokens
```

```
Rule ACTIVITYEDGE_TOKENSFINAL(attrDef, context)
    contextTokens ⇐ context[tokensInitial]                     ▷ query local tokenset
    for all (token : contextTokens) do
        tokensForSplitNode ⇐ tokenMap.get(token.splitNode) ∪ token
        tokenMap.put(token.splitNode, tokensForSplitNode)
        if (tokensForSplitNode→size == token.tokenCount) then
            remainingTokens ⇐ remainingTokens − tokensForSplitNode
    return remainingTokens
```

Three changes have been made compared to the way the original algorithm handles token creation: Instead of an index, the target node is used as a unique identifier for the token (line [11]) as this is a more powerful way of distinguishing between tokens originating from the same gateway. To simplify the converging process, all tokens additionally store the total number of tokens created at the respective gateway [12]. The final change concerns the handling of cycles. Instead of contracting SCCs into single nodes, we have implemented a dedicated handling for this case which makes the token computation aware of cyclic structures: For this reason, token generation is skipped for split gateways inside a SCC [8]. If the gateway is an out port and the SCC has more than one outgoing edge, the sccID is used as split node identifier (line [17]) while the amount of edges leaving the SCC is used to determine the amount of tokens in the tokenset [18].

The rule activityedge_tokensFinal implements the convergence mechanism. The set of available tokens is read from the local context's tokensInitial attribute [2]. A loop then iterates over each token (line [2]), building a map with the split nodes as keys and all tokens generated at these gateways as their values [4 – 5]. If a value entry reaches the size of the outgoing edge count of the split node, this means that all members of the tokenset created at the respective split node have "arrived" at the current edge. In this case, line [7] removes the tokens belonging to this set from the edge's tokenset.

According to the authors of [Got+09], the detection of SESE components inside SCCs is possible if the analysis is repeated for the cycle's inner elements. As preparation, the entering and leaving edges of the cycle must be removed along with the input and output ports. The dangling inner edges formerly linked to the ports are then connected to artificial Initial/FinalNodes. Consequently, an InitialNode is created for each outgoing edge of in ports to act as the edge's new source node. Similarly, each edge formerly associated with an out port is assigned to a newly created FinalNode. The analysis is then repeated for the modified SCC and the results replace the originally computed values for these elements. If the modified cycle again contains a SCC, this process is carried out recursively.

The DFA-based token analysis implements this task using a MAF evaluation macro (cf. Section 8.1.5) which is scheduled for execution after the tokens have been computed. The macro builds a new Activity which contains only the modified SCCs and loads it into the analyzer framework. Finally, the current evaluation strategy is extended with three new directives:

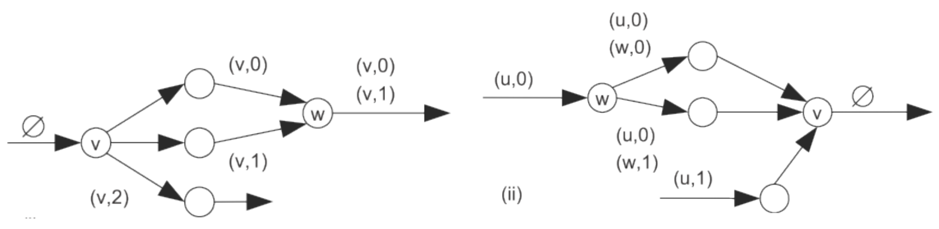
1. An evaluation target that triggers the analysis of the modified SCC.
2. An evaluation macro that transfers the newly computed values back to the result set of the original analysis.
3. The macro recursively adds itself to the end of the list to repeat this process for additional cycles.

To construct the SESE structure tree, components must be identified based on the computed token labelings. The authors of the token analysis algorithm propose a straightforward method for classifying components as cyclic, non-cyclic or sequential (componentType). If exactly two edges exist with the same token labels, they represent the entry and exit of a SESE component. If the component contains a SCC, it is cyclic, otherwise non-cyclic. If the same labeling is available at more than two edges, this denotes a sequence whose first edge (in control-flow direction) is the entry while the last one represents the exit. The hierarchy can also be deduced from the labels: A subcomponent's token labelings fully include its parent's tokens.

The process structure tree can be computed by employing the same method as in the imperative token analysis algorithm. We additionally developed a flow-based prototype as proof-of-concept to demonstrate the applicability of DFA for this purpose. It follows the general premise of collecting tokens along the control-flow and putting them on a stack. On this TokenStack, the components can then be identified using the described properties. Detecting sequences requires additional handling, as only the maximal sequences are of interest. Therefore once-identified sequences must be expanded at successor nodes if it is discovered that more elements exist with the same labeling.

Partial Convergence

[Got+09] describes another useful algorithm called *partial token convergence* which uses the computed token labelings to improve component identification for *quasi-structured* processes. These are characterized by the property that, by simply adding split or merge nodes without changing the execution semantics, additional components can be identified, thereby improving the level of detail of the output.

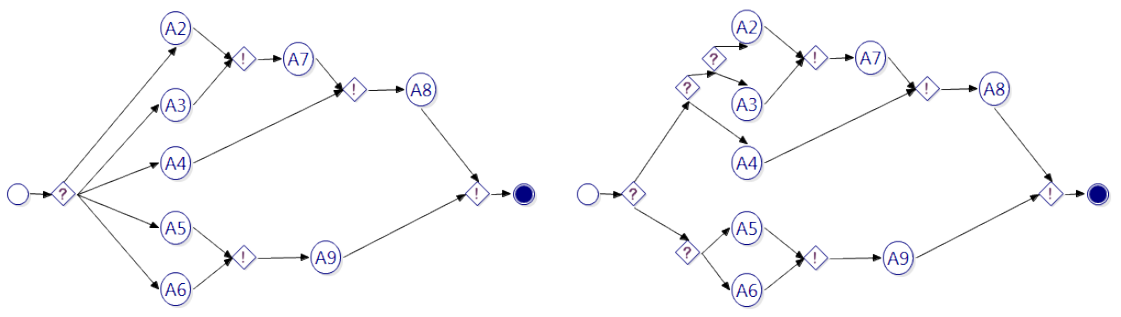


(a) Partial convergence at DecisionNode (b) Partial convergence at MergeNode

Figure 10.13.: Partial token convergence [Got+09].

To implement this functionality, the token analysis uses virtual tokensets consisting of any combination of two tokens in the original tokenset to identify suitable locations for the introduction of new split/merge gateways. In Figure 10.13(a), the tokens $t_{(v,0)}$, $t_{(v,1)}$ and $t_{(v,2)}$ are produced at the outgoing edges of node v. In this case, the virtual tokensets $\{t_{(v,0)}, t_{(v,1)}\}$, $\{t_{(v,0)}, t_{(v,2)}\}$ and $\{t_{(v,1)}, t_{(v,2)}\}$ are generated and propagated along the edges. The presence of the virtual tokenset $\{t_{(v,0)}, t_{(v,1)}\}$ is detected after $\textcircled{w}$. Consequently, a new split gateway $\textcircled{w'}$ can be introduced as a successor of $\textcircled{v}$.

To identify merge nodes for quasi-structured components, virtual tokensets are constructed as partial combinations of the sets received on incoming edges. At the node v shown in Figure 10.13(b), the tokensets $\{t_{(u,0)}, t_{(w,0)}\}$, $\{t_{(u,0)}, t_{(w,1)}\}$ and $\{t_{(u,1)}\}$ are available. By combining two of these sets at a time and after applying token convergence, this step yields the results $\{t_{(u,0)}\}$, $\{t_{(u,1)}\}$ and $\{t_{(w,0)}\}$. The labeling $t_{(u,0)}$ matches the tokenset before node $\textcircled{w}$ meaning that a new merge node $\textcircled{w'}$ can be added before $\textcircled{v}$.



(a) Quasi-structured component (b) Implicit decision nodes have been added

Figure 10.14.: Using partial token convergence to improve component detection.

Again, this algorithm has been implemented using DFA and has been integrated with the analysis specification using a macro call which modifies the underlying model. This is shown in Figure 10.14 which depicts an example process before and after the application of partial token convergence. The second representation, while semantically equivalent to the first, will greatly improve component detection. In the first process, token analysis will only detect a single complex structure while the second process results in a much more fine-grained SESE decomposition.

Structural Validation

Using the method presented in [VVL07], the soundness of (a)cyclic business processes - i.e. the absence of local deadlocks and lack of synchronization - can be tested in linear time. This is accomplished by a bottom-up traversal of the SESE hierarchy, applying heuristics to categorize each sub-graph according to the structure of its elements. If an error is found, the context in which it appears can be used to track down its approximate location, i.e. the (sub)component in which appears.
The authors of [VVL07] define the following criteria:

Let F be a fragment of a workflow graph. F is

1. *well-structured if it satisfies one of the following conditions:*
   - *F has no decisions, merges, forks or joins as children in the process structure tree (sequence),*
   - *F has exactly one decision and exactly one merge, but no forks and no joins as children. The entry edge of F is the incoming edge of the decision, and the exit edge of F is the outgoing edge of the merge (sequential branching),*
   - *F has exactly one decision and exactly one merge, but no forks and no joins as children. The entry edge of F is an incoming edge of the merge, and the exit edge of F is an outgoing edge of the decision (cycle),*
   - *F has exactly one fork, exactly one join, no decisions and no merges as children. The entry edge is the incoming edge of the fork. The exit edge is the outgoing edge of the join. (concurrent branching).*

2. *an unstructured concurrent fragment if F is not well-structured, contains no cycles, and has no decisions and no merges as children.*

3. *an unstructured sequential fragment if F is not well-structured and has no forks and no joins as children.*

4. *a complex fragment if it is none of the above.*

A complex fragment F is not sound if it satisfies one of the following conditions:

1. *F has one or more decisions (merges), but no merges (decisions) as children in the process structure tree,*

2. *F has one or more forks (joins), but no joins (forks) as children,*

3. *F contains a cycle, but has no decisions or no merges as children.*

Since the classification of a component thus relies solely on the (non) existence of certain element types, the computation of these criteria is straightforward as soon as the SESE tree is available. Implementation-wise, this could be achieved by specifying a DFA on the Component model which categorizes each component using the presented properties. The resulting validationClassification provides a more detailed assessment of the respective structural type than the original componentType of the token analysis approach. The soundness of the process (validationResult) can be determined recursively as each component containing unsound children also has to be classified as unsound.

Making use of the capabilities of the Model Analysis Framework, it is also possible to execute this algorithm for the results of a SESE decomposition by realizing it as an evaluation macro. An implementation of the classification algorithm which operates on the results conforming to the metamodel in Figure 10.12 is included in Appendix C.1.2.

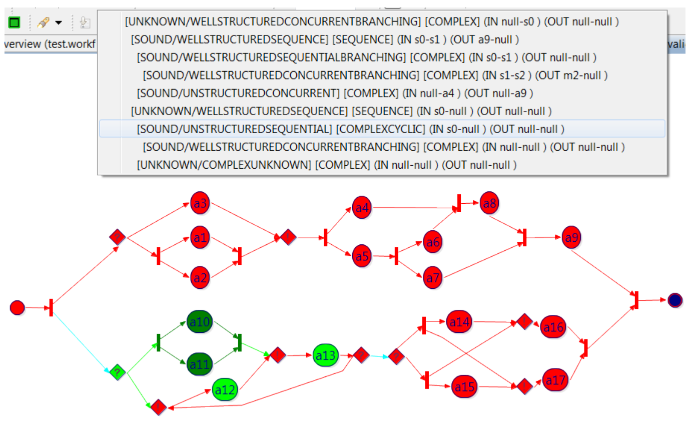

```
[UNKNOWN/WELLSTRUCTUREDCONCURRENTBRANCHING] [COMPLEX] (IN null-s0 ) (OUT null-null )
 [SOUND/WELLSTRUCTUREDSEQUENCE] [SEQUENCE] (IN s0-s1 ) (OUT a9-null )
  [SOUND/WELLSTRUCTUREDSEQUENTIALBRANCHING] [COMPLEX] (IN s0-s1 ) (OUT null-null )
   [SOUND/WELLSTRUCTUREDCONCURRENTBRANCHING] [COMPLEX] (IN s1-s2 ) (OUT m2-null )
  [SOUND/UNSTRUCTUREDCONCURRENT] [COMPLEX] (IN null-a4 ) (OUT null-a9 )
 [UNKNOWN/WELLSTRUCTUREDSEQUENCE] [SEQUENCE] (IN s0-null ) (OUT null-null )
  [SOUND/UNSTRUCTUREDSEQUENTIAL] [COMPLEXCYCLIC] (IN s0-null ) (OUT null-null )
   [SOUND/WELLSTRUCTUREDCONCURRENTBRANCHING] [COMPLEX] (IN null-null ) (OUT null-null )
  [UNKNOWN/COMPLEXUNKNOWN] [COMPLEX] (IN null-null ) (OUT null-null )
```

Figure 10.15.

Figure 10.15 shows the example process from [VVL07]. It has been modeled in JWT and subjected to the structural validation (yielding the same results as in the original paper). The box in the upper region of the screenshot contains the classified SESE tree. Each entry represents a (sub)component and lists its soundness (validationResult) and classification (validationClassification) property as well as the type assigned by the tokenflow analysis (componentType) and the respective in and out edges.

Translation to BPEL

The final use case which depends on the hierarchical SESE decomposition maps graph-oriented business process modeling languages such as JWT or BPMN to executable block-oriented representations such as the Business Process Execution Language (BPEL). Transforming the models between these different representations while preserving semantics is an inherently difficult task. Graphs allow for a more complex control-flow while block-oriented languages employ constructs typically found in imperative programming languages such as loops and therefore adhere to a more rigid structure.

Several solutions have been proposed for handling the difficulties which arise during the translation process. The approaches presented in [Gar08; MLZ06] are of special interest in the context of this thesis. They make use of the SESE process tree and data-flow analysis to determine BPEL mappings for fragments of BPMN processes which can then be translated using a template-based approach. As JWT shares the properties of BPMN which are relevant to the application of this method, we can make the assumption that the following discussion also applies to the models presented in this case study.

The author of [MLZ06] defines a set of data-flow equations which can be applied to each identified SESE region. These compute information about transitive predecessors (for whole regions as well as limited to sequences), the amount of preceding split nodes etc. From these results, mappings can be derived for the translation of the components into corresponding BPEL constructs. A more detailed listing of relevant BPEL structures and their respective counterparts in BPMN can be found in [ODA08]. Just like with the validation technique presented above, parts of this analysis can be implemented as a macro which operates on the results of the SESE decomposition and stores the results in the generated Component model.

10.1.5. Use Case: Model Clone Detection

It is often desirable to avoid replicating the same (or similar) structures. This principle applies to both program code as well as to models. For this reason, the detection of so-called clones, i.e. substructures that resemble each other according to a predefined set of properties, is an important research area. According to [Kos06], *"code clones are fragments that are similar w.r.t. to some definition of similarity"*. [RCK09] notes that, while cloning in software systems may be intentional, *"it can also be harmful in software maintenance and evolution"*. The presence of clones has an effect on the detection and removal of bugs, the effort for enhancing or adapting code and complicates tasks such as program understanding, code compaction etc.

The latter paper describes a categorization for code clones and performs an in-depth survey of existing approaches and categorizes them according to different characteristics.

[ADS12] states that *"while its counterpart, code clone detection, is a mature and established area of research, model clone detection is relatively new and has not been investigated as thoroughly"*. Because *"unlike source code, which is represented as linear text, models are typically represented visually, as box-and-arrow diagrams, [. . .] the clones we are searching for are similar subgraphs of these diagrams"*. [SC13] contains a survey of different approaches and tools for model clone detection operating on domains ranging from MATLAB Simulink models to UML Sequence Diagrams, Statecharts and structural models, business process models and also metamodel-agnostic techniques.

The importance of clone detection in the context of the modeling of business processes is stressed by [Uba+11]. Multiple reasons are given to explain why clones have a negative impact on the tasks of process management and maintenance: *"Clones make individual process models larger than they need to be, thus affecting their comprehensibility. Secondly, clones are modified independently, sometimes by different stakeholders, leading to unwanted inconsistencies across models that originally contained a duplicate clone. Finally, process model clones hide potential efficiency gains. Indeed, by factoring out cloned fragments into separate subprocesses, and exposing these subprocesses as shared services, companies may reap the benefits of larger resource pools"*.

Three types of model clones are defined by [ADS12] which are similar to the categories for code clones listed in [Kos06]:

Exact model clones Model fragments that only differ in visual presentation, layout and formatting.

Renamed model clones Structurally identical, except for variations in labels, values, types, visual presentation, layout and formatting.

Near-miss model clones Additionally allows for differences such as change in position or connection with respect to other model fragments and small additions or removals of blocks or lines.

Both the first and the second clone type rely on structurally identical (isomorph) subgraphs. The second type employs a similarity measure to determine whether two elements correspond to each other. This allows for certain variations, e.g. to compensate for spelling errors in labelings. In this use case, we realize a flow-based method for the detection of exact and renamed model clones which has been inspired by the ConQAT algorithm. Some elements of this mechanism have been developed in the context of the master thesis [Kra12].

The ConQAT Algorithm

We base our implementation on the problem definition and the terminology introduced by [Dei+08]. This paper applies the ConQAT algorithm to detect clones in

Simulink models. The measurements provided for this method will also serve as a baseline for the evaluation of the performance of the DFA-based approach.

Clone pairs are subgraphs which are considered to be clones with the following definition:

A clone pair is a pair of subgraphs (V_1, E_1), (V_2, E_2) *with* $V_1, V_2 \subset V$ *and* $E_1, E_2 \subset E$*, such that the following conditions hold:*

1. *There are bijections* $\wr_V : V_1 \longrightarrow V_2$ *and* $\wr_E : E_1 \longrightarrow E_2$*, such that for each* $v \in V_1$ *it holds* $L(v) = L(\wr_V(v))$ *and for each* $e = (x, y) \in E_1$ *it is both* $L(e) = L(\wr_E(e))$ *and* $(\wr_V(x), \wr_V(y)) = \wr_E(e)$.
2. $V_1 \cap V_2 = \emptyset$
3. *The graph* (V_1, E_1) *is connected.*

For $(V_1, V_2) \subset V$*, we say that they are in a cloning relationship, iff there are* $(E_1, E_2) \subset E$ *such that* (V_1, E_1)*,* (V_2, E_2) *is a clone pair.*

This definition is summarized as follows: *"The first condition of the definition just states that those subgraphs must be isomorphic regarding to the labels L, the second one rules out overlapping clones, and the last one ensures we are not finding only unconnected blocks distributed arbitrarily through the model".*

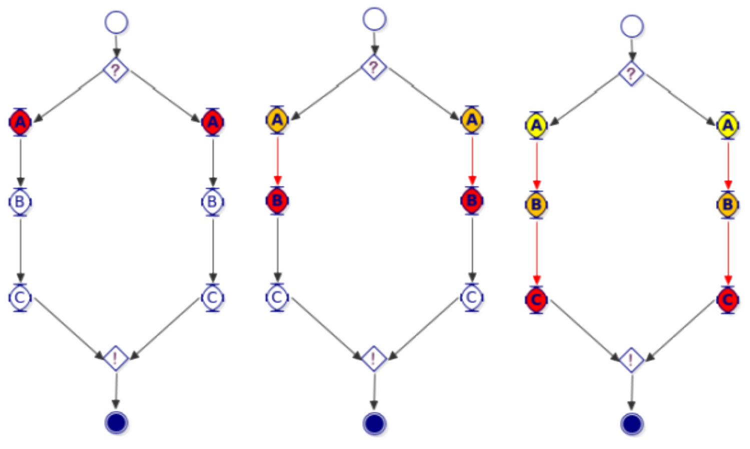
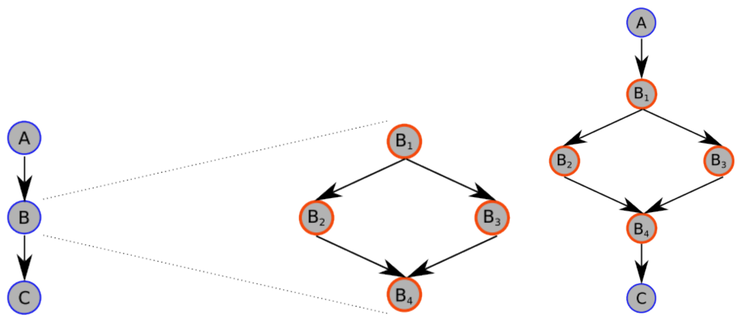


Figure 10.16.: Iterative detection of maximal clones [Kra12].

Figure 10.16 demonstrates the expansion of a clone pair with the initial size of one to its final extent of three nodes. In this case, this value represents the maximal size as adding either the DecisionNode or the MergeNode would lead to overlapping clones. A major challenge of clone-detection methods in general is mentioned in [Dei+08]: The problem of finding maximal clone pairs is NP-complete[17] meaning that no algorithm exists which can detect all maximal clones in polynomial time.

[17]It is stated that the detection of maximal clones inside a model graph is very similar to the NP-complete Maximum Common Subgraph (MCS) problem.



(a) Structured model

(b) Model after inlining

Figure 10.17.: Applying inlining to structured models [Kra12].

The clone detection must be preceded by a normalization step. This process includes the inlining of substructures. By copying and inserting embedded or linked activities, the model graph is flattened. This approach can be compared to the transformation of LinkedActivityNodes into StructuredActivityNodes as required by the previous use cases and is exemplified in Figure 10.17. Typically, it is expected that the data structure which represents the input graph $G = (V, E, L)$ not only consists of vertices V and edges E but also defines labelings L for these elements. The labelings are computed by the function $L : V \cup E \to N$. By customizing this function, node and edge labelings can be normalized[18]. Since this process influences the circumstances under which elements are considered to be clones of each other, this step can be used to implement a detection of renamed model clones.

Algorithm 34 The ConQAT algorithm [Dei+08]

```
 1: $D = \emptyset$
 2: for all $(u, v) \in V \times V$ with $u \neq v \wedge L(u) == L(v)$ do
 3:     if $\{u, v\} \notin D$ then
 4:         Queue $Q \leftarrow \{(u, v)\}$, $C \leftarrow \{(u, v)\}$, $S \leftarrow \{u, v\}$
 5:         while $Q \neq \emptyset$ do
 6:             dequeue pair $(w, z)$ from $Q$
 7:             from the neighborhood of $(w, z)$ build a list of node pairs $P$ for which the
    conditions (1,2) hold
 8:             for all $(x, y) \in P$ do
 9:                 if $(x, y) \in D$ then
10:                     continue at line 2
11:                 if $x \neq y \wedge \{x, y\} \cap S == \emptyset$ then
12:                     $C \leftarrow C \cup \{(x, y)\}$, $S \leftarrow S \cup \{x, y\}$
13:                     enqueue $(x, y)$ in $Q$
14:     report node pairs in $C$ as clone pair
15:     $D \leftarrow D \cup C$
```

[18]Normalization may include stemming of labels (or manual specifications of equality) and may respect or dismiss additional information such as class types or edge guards.

The ConQAT approach is outlined in Algorithm 34. This method iterates over all possible node pairings and processes them if both nodes possess the same labels [2]. The steps in lines [4−13] correspond to a breadth-first search which is used to explore the nodes' neighborhoods. *"During this [...] the sets C of current node pairs in the clone, S of nodes seen in the current BFS, and D of node pairs we are completely done with"* are managed. Line [7] applies a weighted similarity function which decides whether two nodes are similar based on their respective neighborhoods. To reduce the number of necessary comparisons only a single mapping is considered. In the final step, the identified clone pairs are clustered into clone classes depending on the contained nodes while ignoring differences in the sets of edges to allow for a certain flexibility.

Flow-based Clone Detection

We will now study a flow-based implementation of a clone detection algorithm. The stated goal here is the detection of exact and renamed model clones and the prevention of the detection of overlapping subgraphs in a way that is comparable to the presented ConQAT method.

The developed algorithm relies on the computation of the shortest paths connecting each node to the start node. While this information is propagated throughout the target graph, partial representations of these paths are built. By comparing the result sets, initial candidates for (sequential) clone structures can be identified. The clones are then expanded by adding nodes surrounding these base graphs as long as they possess the same structural layout.

Algorithm 35 Data-flow rule for computing the shortest paths

```
1:  Rule ACTIVITYNODE_SHORTESTPATHS(attrDef, context)
2:      for all (predNode : context.in.source) do                ▷ process predecessors
3:          predShortestPaths ⇐ predNode[shortestPaths]
4:          predResult ⇐ ∅
5:          for all (shortestPath : predShortestPaths) do         ▷ incoming paths
6:              if (shortestPath→contains(context)) then
7:                  continue at [2]                               ▷ abort at back edges
8:              predResult ⇐ predResult ∪ (shortestPath→prepend(context))
9:          result ⇐ predResult ∪ (predNode→prepend(context))
10:     return result
```

For a fully connected and normalized graph, the algorithm depends on the following steps:

1. In the first step, we apply DFA to derive the set of partial (cycle-free) paths P_N of length > 1 connecting each node N to the start node. For a model Ⓢ $\rightarrow$ Ⓐ $\rightarrow$ Ⓑ (with Ⓢ being the start node), $P_B = \{$Ⓑ $\leftarrow$ Ⓐ, Ⓑ $\leftarrow$ Ⓐ $\leftarrow$ Ⓢ$\}$.

   The computation of P_N is carried out as follows: First, the results at incoming paths, i.e. the values calculated at each predecessor node $N_{pred} \in$

$pred(N)$, are requested. This set is then extended with the predecessor nodes themselves (or rather, paths of length 1 consisting only of the predecessors): $\bigcup_{pred=predecessor(N)}(P_{pred} \cup pred)$. Now, node N is added as a prefix to the beginning of each of these paths and the result is returned as P_N.

For the correct handling of cyclic paths, it is essential that the execution stops at back edges. This can be achieved by checking if the current node is already included in a path in N_{pred} in which case the input from this predecessor is ignored. The data-flow rule which carries out this computation is shown in Algorithm 35.

2. Next, the sets of partial paths are identified and clustered to prepare for the subsequent detection of maximal clones. This happens by comparing the label of each node to the labels of all other nodes which are part of the model. If a group of similar nodes $N_1 \ldots N_i$ has been identified, their respective P_N sets are grouped together in the clone group candidate $CGC_{N_1 \ldots N_i}$.

   This structure forms the basis for the detection of clone candidates and in many cases substantially reduces the required number of comparisons. It should be noted that clones which consist only of sequential nodes are already fully contained in this set as paths of the same length.

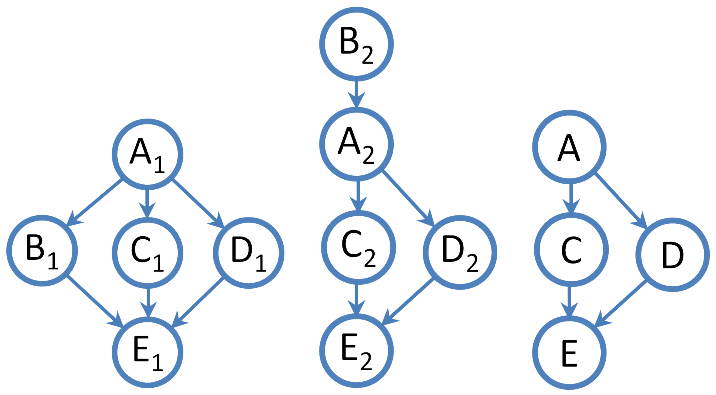
3. The sets of clone group candidates CGC are then processed one by one. Path entries originating at one node of a matching set are compared to paths of all other nodes of the same length starting with the longest candidate. If a path matches one or more other paths with no overlap between them, a new clone candidate is created and all other paths of the same CGC which are partial representations of the matched graph can be removed. Because of the no-overlap condition, CGC sets for nodes for which a clone relationship has already been detected do not need to be processed subsequently and can therefore be ignored. If a clone candidate overlaps with another clone candidate, the two are merged. This way, the clone candidates are extended incrementally so that after the algorithm has finished, they contain the maximal amount of matching nodes, i.e. the clone groups.

Examples

We demonstrate the application of the DFA-based clone detection using the examples shown in Figure 10.18(a) and Figure 10.18(b). While normally, these subgraphs would be part of the same model, for reasons of clarity we have separated them into two different structures. Nodes which differ only in their index are assumed to represent matching elements according to the labeling function L. $\textcircled{A_1}$ and $\textcircled{A_2}$ therefore represent a clone pair of the size one. We assume that the normalization step has already been carried out.

In the first step, the sets of (reversed) partial paths P_N are computed for each node. The results are shown in Table 10.5.

It is now possible to derive the clone groups candidates CGC by comparing the nodes. The results of this process are shown in Table 10.6. Clones inside this



(a) Graph 1 (b) Graph 2 (c) Clone

Figure 10.18.: Graphs containing a clone structure.

Node	*Shortest paths* (P_N)
A_1	
B_1	$(B_1 \leftarrow A_1)$
C_1	$(C_1 \leftarrow A_1)$
D_1	$(D_1 \leftarrow A_1)$
E_1	$(E_1 \leftarrow B_1)$, $(E_1 \leftarrow B_1 \leftarrow A_1)$, $(E_1 \leftarrow C_1)$, $(E_1 \leftarrow C_1 \leftarrow A_1)$, $(E_1 \leftarrow D_1)$, $(E_1 \leftarrow D_1 \leftarrow A_1)$
A_2	$(A_2 \leftarrow B_2)$
B_2	
C_2	$(C_2 \leftarrow A_2)$, $(C_2 \leftarrow A_2 \leftarrow B_2)$
D_2	$(D_2 \leftarrow A_2)$, $(D_2 \leftarrow A_2 \leftarrow B_2)$
E_2	$(E_2 \leftarrow C_2)$, $(E_2 \leftarrow C_2 \leftarrow A_2)$, $(E_2 \leftarrow C_2 \leftarrow A_2 \leftarrow B_2)$, $(E_2 \leftarrow D_2)$, $(E_2 \leftarrow D_2 \leftarrow A_2)$, $(E_2 \leftarrow D_2 \leftarrow A_2 \leftarrow B_2)$

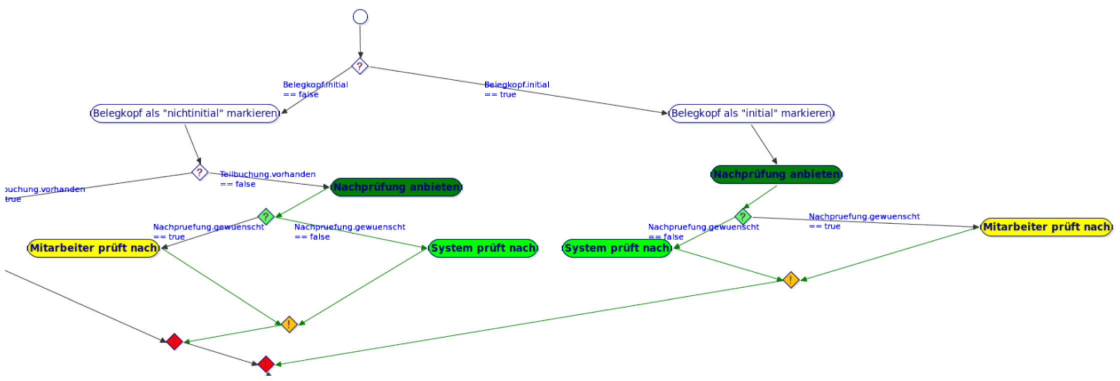
Table 10.5.: Shortest paths to the start node.

structure can then be identified by searching for matching paths of the same length which do not overlap. For example, the path $(E_1 \leftarrow C_1 \leftarrow A_1)$ computed at $\textcircled{E_1}$ is compared to $(E_2 \leftarrow C_2 \leftarrow A_2)$ and $(E_2 \leftarrow D_2 \leftarrow A_2)$ which represent the results for the matching node $\textcircled{E_2}$. This results in the initial clone candidate $(A_{1/2}, C_{1/2}, E_{1/2})$. As a consequence, the partial paths $(E_1 \leftarrow C_1)$ and $(E_2 \leftarrow C_2)$ can now also be removed. A further comparison of $(E_1 \leftarrow B_1 \leftarrow A_1)$ and $(E_1 \leftarrow D_1 \leftarrow A_1)$ with the same paths of $\textcircled{E_2}$ yields another initial clone $(A_{1/2}, D_{1/2}, E_{1/2})$. Again, the contained partial paths $(E_1 \leftarrow D_1)$ and $(E_2 \leftarrow D_2)$ are removed since they already belong to a clone. Because both initial clones overlap, they are combined, resulting in the final clone pair $V_1 = (A_1, C_1, D_1, E_1)$, $V_2 = (A_2, C_2, D_2, E_2)$ (shown in Figure 10.18(c)). Because overlapping between clones is not allowed, we can remove $CGC_{A_{1/2}}$, $CGC_{C_{1/2}}$ and $CGC_{D_{1/2}}$ from the list of sets which must still be processed.

Figure 10.19 shows the application of the clone detection on a real world example,

CGC	*Clone candidate paths*
$A_{1/2}$	$A_2 : \{(A_2 \leftarrow B_2)\}$
$B_{1/2}$	$B_1 : \{(B_1 \leftarrow A_1)\}$
$C_{1/2}$	$C_1 : \{(C_1 \leftarrow A_1)\}$ $(C_2 : \{(C_2 \leftarrow A_2), (C_2 \leftarrow A_2 \leftarrow B_2)\}$
$D_{1/2}$	$D_1 : \{(D_1 \leftarrow A_1)\}$ $(D_2 : \{(D_2 \leftarrow A_2), (D_2 \leftarrow A_2 \leftarrow B_2)\}$
$E_{1/2}$	$E_1 : \{(E_1 \leftarrow B_1), (E_1 \leftarrow B_1 \leftarrow A_1), (E_1 \leftarrow C_1), (E_1 \leftarrow C_1 \leftarrow A_1),$ $(E_1 \leftarrow D_1), (E_1 \leftarrow D_1 \leftarrow A_1)\},$ $E_2 : \{(E_2 \leftarrow C_2), (E_2 \leftarrow C_2 \leftarrow A_2), (E_2 \leftarrow C_2 \leftarrow A_2 \leftarrow B_2),$ $(E_2 \leftarrow D_2), (E_2 \leftarrow D_2 \leftarrow A_2), (E_2 \leftarrow D_2 \leftarrow A_2 \leftarrow B_2)\}$

Table 10.6.: Clone group candidates containing identified shortest paths.



Belegkopf.initial == false
Belegkopf.initial == true
Belegkopf als "nichtinitial" markieren
Belegkopf als "initial" markieren
Teilbuchung.vorhanden == false
ɔuchung.vorhanden true
Nachprüfung anbieten
Nachprüfung anbieten
Nachpruefung.gewuenscht == true
Nachpruefung.gewuenscht == false
Mitarbeiter prüft nach
System prüft nach
System prüft nach
Nachpruefung.gewuenscht == false
Nachpruefung.gewuenscht == true
Mitarbeiter prüft nach

Figure 10.19.: Clone detection using DFA [Kra12].

in this case an excerpt of a business process taken from a SAP system and modeled in JWT. The full example can be found in Appendix C.1.3.

Conclusions and Outlook

The identification and extraction of identical (or similar) subprocesses reduces maintenance efforts and provides a better overview over the implemented functionality. The application of this method therefore enables an improvement of existing processes and process libraries. The presented approach for flow-based clone detection serves as proof that the DFA approach can provide valuable input for the implementation of complex algorithms.

The described algorithm facilitates the detection of initial clone candidates in the form of paths which are subsequently extended in an incremental fashion. Flow analysis is employed to compute combinations of shortest paths connecting each node to the start node. The paths sets are then compared to each other to identify suitable paths which represent the backbones of potential clones. These are then extended to full clone graphs.

The algorithm provides several optimizations which reduce the number of required comparisons: For example, only paths of length > 1 are subjected to clone detection. Because, by definition, clones must not overlap, *CGC* sets for nodes which have already been classified as a partial clones can be discarded immediately, thereby greatly reducing the amount of necessary comparisons. Furthermore, only paths of the same length are matched.

We can state that, just like the ConQAT algorithm, we are able to correctly detect structural and renamed clones as long as the target graphs are isomorphic. Another property which is shared by both approaches is the disregarding of overlapping clones. In its current version, our method is however not able to identify near-miss clones. Performance-wise, the DFA overhead of the path generation carried out in the first step is typically negligible for small to medium sized models since the size and the amount of paths depends linearly on the number of predecessors and no fixed-point reevaluation is required. The main performance impact stems from the path comparisons during the last step. As a consequence, the actual complexity depends heavily on the layout of the graph. This is however a problem which is inherent to all clone detection mechanisms.

The presented method is only an initial exploration of the concept of implementing a method for clone detection which is supported by DFA. Consequently, there are many conceivable ways to improve the described approach:

- The identification of back edges in the initial step can be sped up using an analysis which keeps track of "visited" nodes. This could be interpreted as a DFA implementation of a depth-first search algorithm.

- The grouping of paths P in clone path candidates in the second step works especially well if the sets of matching nodes are small, i.e. if their labelings are distinct. If this is not the case, e.g. when searching for structural clones, the creation of the CGC sets can be assisted by factoring in the structure of the immediate neighborhood of the nodes into the labeling function.

- The CGC sets can be represented as trees. This could be used to speed up the elimination of partial paths if an encompassing clone has been detected.

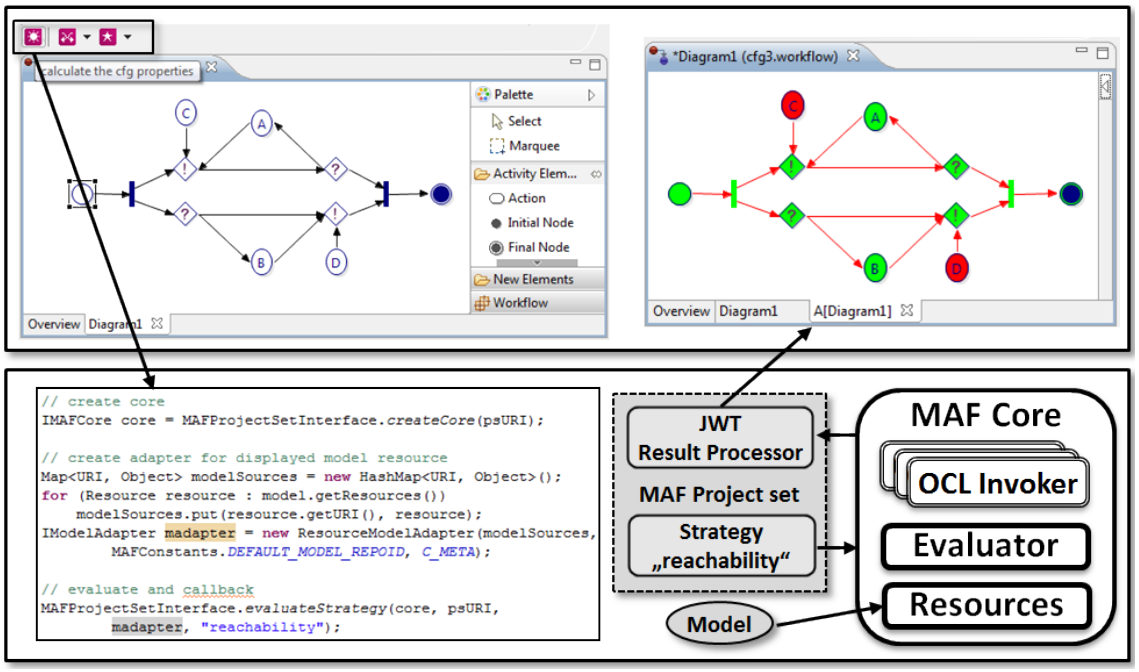
- Just like in ConQAT, we can limit the clone detection to a single candidate. This means that that if a path $(A_2 \rightarrow D_3 \rightarrow E_2)$ existed in Figure 10.18(b), it would be disregarded as a possible clone in case the clone has already been extended by the path $(A_2 \rightarrow D_2 \rightarrow E_2)$, containing the "conflicting" node $\textcircled{D_2}$.

- The SESE decomposition presented in Section 10.1.4 could be used to cluster the target graph into larger components. A subsequent clone detection could then make use of this information to speed up the identification of similar regions by focusing on the SESE structures.

- Finally, DFA could be applied to potential clones as early as possible. Consider the graphs in Figure 10.18(a) and Figure 10.18(b): Assuming the existence of paths $(E_1 \rightarrow F)$ and $(E_3 \rightarrow F)$ which connect both graphs to a gateway F, the clone relationship $(A_{1/2}, C_{1/2}, D_{1/2}, E_{1/2})$ becomes evident at F. This way, all partial paths which are contained in this clone could be removed, thereby reducing the number of comparisons in subsequent steps.

10.1.6. Tooling Integration

Integration of the analyses with the JWT framework is facilitated through Eclipse's plugin mechanism. Since the required functionality for accessing model data and visualizing the results is accessible via the API provided by JWT, the tooling did not have to be modified in any way[19].



```
// create core
IMAFCore core = MAFProjectSetInterface.createCore(psURI);

// create adapter for displayed model resource
Map<URI, Object> modelSources = new HashMap<URI, Object>();
for (Resource resource : model.getResources())
    modelSources.put(resource.getURI(), resource);
IModelAdapter madapter = new ResourceModelAdapter(modelSources,
        MAFConstants.DEFAULT_MODEL_REPOID, C_META);

// evaluate and callback
MAFProjectSetInterface.evaluateStrategy(core, psURI,
        madapter, "reachability");
```

Figure 10.20.: Integrating analysis capability into JWT.

For this case study, a customized ModelAdapter has been implemented which is responsible for loading and normalizing the business process models. For this purpose, it converts linked Scopes (LinkedActivityNodes) to embedded activities (StructuredActivityNodes), thereby duplicating referenced elements for each context in which they appear. To track which object in the result set corresponds to which element in the original model, a trace map is built during this process. Each element is uniquely identified by a path derived from its containment hierarchy.

Because of the large number of analysis use cases for JWT models, a common framework has been established which acts as a bridge between the analysis components and the target application. This technology bridge (cf. Section 8.3.1) handles the initialization of MAF, the loading of metamodel and model data and implements shared functionality for result visualization. Adding a new flow analysis therefore only requires the specification of the analysis itself as well as a definition of how the results should be interpreted, e.g. which color should be assigned to a model element based on its respective data-flow result. The overall concept is outlined in Figure 10.20: Based on the available analyses, a set of commands is dynamically added to the main toolbar of the application window. Invoking one of these options

[19]The implementation of the use cases is available from the official MAF repositories.

triggers the technology bridge which acquires the currently selected model from the JWT workflow editor. The model is then normalized and loaded into MAF using the customized model adapter. In the next step, the chosen analysis is executed using the Project Set interface. Finally, a visualization is generated for the respective analysis results. Using the toolbar commands, it is now possible to switch between different result representations which are immediately displayed in the graphical editor.

10.1.7. Evaluation of the Use Cases

Section 10.1.2 describes how different kinds of control-flow analyses such as reachability and flowset computation can be adapted to the domain of business process modeling. To evaluate the scalability of the fixed-point computation for models, we employed the built-in debugging facilities of MAF (cf. Section 8.3.4) to record performance data. This evaluation therefore mirrors the steps carried out in Section 6.5.4 to assess the properties of the dependency chain algorithm using the control-flow modeling language which served as running example.

Because of the required fixed-point iterations and due to the large number of results produced by the DFA approach, it is not surprising that, with respect to execution time and memory consumption, DFA-based implementations have a larger overhead when compared with algorithms which are tailored to solve a specific problem such as Tarjan's algorithm for SCC detection [Tar72]. However, the flow analysis approach provides a multi-purpose "analysis programming language" which enables concise, declarative analysis specifications which directly reflect the propagation paths of information and also supports a combination of analyses to implement more complex scenarios. For example, traditional flow analysis for finding dominators (allPredecessorsMin) runs in $\mathcal{O}(n^2)$ [CHK01], although different algorithms have been proposed that run faster [Geo+04]. It can be assumed that, for the average case, DFA-based implementations perform much better than in the worst case scenario as the runtime is highly dependent on the amount of contained cycles. Therefore, one has to weigh the performance advantage of a more complex, proprietary solution against the straightforward specification enabled by data-flow analysis.

A prerequisite for executing the tokenflow analysis presented in Section 10.1.4 is the computation of allPredecessorsMax, sccID and sccOutEdges. The analysis itself however does not produce new information inside SCCs, so both the token propagation and token convergence steps only require a single iteration.

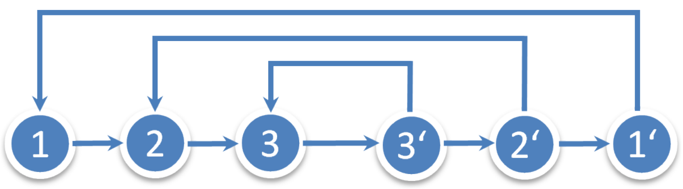


Figure 10.21.: Worst case for tokenflow analysis.

In the case of nested SCCs, the evaluation has to be repeated recursively for cyclic subgraphs. The worst case scenario is depicted in Figure 10.21. Here, only one cycle can be removed after the tokens have been computed which means that the new graph is only 2 nodes smaller than the original one. In this configuration, initial and final tokensets must be calculated for models containing six, then four and finally two nodes. The total amount of necessary computations in relation to the nodes therefore amounts to $2 \times \sum_{k=1}^{(n/2)} 2 \times k$ or $\frac{n^2}{2} + n$. Since the number of components is always smaller than the amount of nodes and each component consists of at least two nodes, there can be only $\frac{n}{2}$ components. Therefore, the detection and classification depends on the performance of the used storage mechanism in relation to this worst case number.

As expected, with about 180 lines of code the DFA variant of the tokenflow analysis is much more compact than the original reference implementation[20] which implements the token generation in about 400 lines of code with additional 300 lines for handling SCCs[21]. The need for assigning time-stamps to nodes to identify their relative position in the control-flow and the usage of the visitor pattern to process the nodes also hides the inherent data-flow properties of this algorithm.

In Section 10.1.5, we demonstrated how a DFA-based method can be used to detect clones in (business process) models. The chosen method was inspired by the ConQAT algorithm which identifies similar nodes and expands matching candidate sets with additional nodes if their neighborhood is similarly structured to yield maximal, non-overlapping clone pairs. Performance-wise, clone detection tends to be a challenging task. [Dei+08] states that for ConQAT *"the time and space requirements for the clone pair detection are depending quadratically on the overall number of blocks in the model"*. The described algorithm represents an initial attempt at DFA-supported clone detection and still requires an in-depth study of its computational complexity. First tests carried out on real-world examples indicate that it performs reasonably well for medium-sized models and the optimizations proposed in Section 10.1.5 may help to further improve the performance.

10.1.8. Summary and Discussion

In this case study, we focused on several use cases to demonstrate the applicability of static analysis in the business process modeling domain. JWT was chosen as a basis for these implementations since it is conceptually and technically rooted in model-driven principles. Furthermore, its similarities to the widely-used BPMN standard, the extensibility of the Eclipse-based tooling and the inherent control-flow structure of JWT business process models make it a feasible candidate for DFA both on a theoretical and on a practical level.

The first use case (cf. Section 10.1.2) realizes the computation of many traditional DFA properties by adapting these analyses to the BPM domain. The examples described in the subsequent sections detail additional methods for the evaluation of business processes which provide valuable feedback to developers and can also

[20] `http://tokenanalysis.svn.sourceforge.net`

[21] Not including datatypes for storing tokens and tokensets as well as internal graph representations.

be used as input for other analyses. The latter property is especially evident in the incrementally structured SESE use case (cf. Section 10.1.4) which employs already computed information to generate more sophisticated results. The tokenflow analysis method represents a reimplementation of an existing, imperative algorithm which allows for a direct comparison between both approaches.

The control-flow properties are easily adapted to this domain because business processes resemble directed graphs. While these results can be useful on their own, e.g. by giving an indication which steps will be executed before a certain point in the workflow is reached, they are also a prerequisite for many of the subsequent analyses. The definition/usage scenario laid out in Section 10.1.3 extracts information about the availability status of resources at different points in the execution of process instances and therefore not only validates the structural composition of process models but also conveys information about the status of resources that can help to understand the semantics of the modeled workflow. The token analysis from Section 10.1.4 generates and propagates tokens to identify SESE regions. This use case serves as an example for flow-based reimplementations of existing approaches and demonstrates how the proposed method can be used to combine different algorithms to build an integrated solution for the decomposition, validation and transformation of the input models. It also shows that data-flow analysis - which normally operates on flat value sets - can be used to generate more sophisticated results (such as SESE components), relying on domain-specific datatype metamodels. Finally, Section 10.1.5 presented an existing clone detection algorithm along with an implementation which employs results derived using DFA. Although the flow-based algorithm still requires a thorough evaluation with respect to its performance, this use case demonstrates how flow analyses can be used to generate intermediate results that can later be used as input for other algorithms. In some cases, DFA-based implementations may suffer from a heavier performance impact when compared to algorithms which are tailored for solving a very specific problem. However, the generality of the approach makes it a viable alternative to quickly implement algorithms (or parts thereof) that rely on information propagation. Overall, it can be stated that many methods which operate on graph structures can be formalized as a flow analysis by making use of the graph's inherent flow properties.

In summary, we can make the following observations:

Implementation effort

JWT as well as BPMN share similar concepts and are built on OMG's metamodeling framework. As such, the integration of flow-based model analysis is a straightforward task although domain-specific adjustments are necessary (e.g. converting linked activities to embedded subprocesses). Making use of the capabilities provided by the modeling framework EMF and the analysis tool MAF this is easily accomplished. Because JWT is based on Eclipse, its modeling and visualization facilities are accessible through the provided APIs. In fact, no pre-existing code had to be modified to implement the analyses and the corresponding result visualizations. The analyses themselves make use of parameterized methods to avoid the duplication of functionality and reduce the overall efforts for implementation and maintenance tasks.

Classification
It was shown that model-based data-flow analysis not only enables the computation of traditional flow-based properties but also supports the derivation of domain-specific information such as data/resource availability (cf. Section 10.1.2). The token analysis use case presented in Section 10.1.4 demonstrates that DFA actually simplifies the implementation of algorithms that rely on information propagation based on declarative specifications rather than using a more tedious and counterintuitive imperative approach. Combined with validation and transformation tasks which have been implemented using MAF macros, this use case also demonstrates the feasibility of using DFA to build an integrated solution which supports the user by offering a collection of related functions.

Reuse
The control-flow analyses in Section 10.1.2 are heavily influenced by the general methods described in Section 9.2. These templates have been adapted for the BPM domain and extended with the capability to examine the contents of subprocesses. The information which is computed by these analyses is then reused in other scenarios. For example, properties of SCCs are an integral part of the token analysis described in Section 10.1.4.

Usability
From a user's perspective, the tight integration with the existing JWT tool chain means that it is not necessary to switch contexts when executing an analysis or examining the results. Also, developers are able to benefit from easy access to the APIs of the analysis tooling and the modeling facilities.

Performance
Several steps have been taken to improve the performance of the algorithms including the usage of bitvector types and the employment of the data-flow initialization constant (*INIT*) to avoid the computation of the value domain. A qualitative evaluation has shown that - even for large process models - the analyses perform reasonably well, making their application in every day usage scenarios feasible.

Multiple starting points for improvement have been identified. For example, the evaluation of data availability may benefit from a backward flow analysis to improve the assessment of the status of unused objects. The section about token analysis and SESE decomposition describes how the results can be used to transform graph-oriented to block-oriented languages. Furthermore, by computing alternative paths connecting an arbitrary node to the start node, the information conveyed by the predecessor analysis could be vastly improved. The result would not only determine which actions may have been executed before a specific point in the process but also indicate all unique (cycle-free) paths leading from the start node to this action. Furthermore, common metrics such as cohesion or complexity could be calculated for process models to assess their quality [Van+07]. Based on estimates made by domain experts about the time required for the execution of business actions, the minimal and maximal execution time for the whole process could be approximated. Finally, if only a limited amount of specific resource types is available, DFA can be used to approximate the maximum number of concurrent accesses to a resource on (nested) parallel paths. This information can be used to identify potential bottlenecks in the process design.

10.2. Case Study: Enterprise Architecture Management

In this case study, we will present two use cases in the context of Enterprise Architecture Management (EAM). As the IT landscape in organizations becomes more and more complex, many companies face the challenge of implementing an efficient and reliable management of their business-critical systems. The research field of Enterprise Architecture Management intends to improve this situation by offering methods and tools which enable organizations to model the technical and business-oriented properties of their IT systems. [Lan12] notes that EAM supports organizations in getting a clear understanding of their essential business and IT elements. This understanding can be further improved by analyzing the modeled EA data, thereby helping enterprise architects to assess different aspects of the company's IT infrastructure. Analysis can be a valuable tool for the identification of potentially problematic situations such as bottlenecks in the allocation of business services and resources. Furthermore, an evaluation of the current landscape allows organizations to develop a goal-oriented IT strategy for the improvement of the reliability and availability of the services provided by their computing infrastructure.

The research work presented in this section is based on two publications:

- [LSB14a] develops an approach for the computation of *key performance indicators* (KPI) based on a generic model-based representation of EAM data.
- [LSB14b] covers the topic of *impact analysis* which enables enterprise architects to examine how the change of an element (e.g. the failure or replacement of a server) will affect the remainder of the organization's IT-based services.

Both methods are based on a flow-sensitive analysis of the underlying EAM models and have been realized using the Model Analysis Framework. The contents of this section represent an adapted version of the original publications.

10.2.1. Introduction and Motivation

Enterprise Architecture Management provides methods for the management of the large IT infrastructures encountered in today's organizations. As a result, models in the EA domain often contain a large amount of elements which are connected through complex relationships. To benefit from this methodology once it has been successfully established in an organization, it is therefore vital to employ suitable methods which support the (semi-)automatic analysis of EA data. [Joh+07b] describes the analysis of EAM models as the *"application of property assessment criteria on enterprise architecture models"*.

The use of analysis functionality to leverage the representations of IT infrastructures has many possible applications. An important reason for the relevance of analysis in this domain stems from the fact that architectural models *"provide only visual and qualitative support"* [FFJ09] and EA analysis is therefore required to quantify these models. It can, for example, be used to support decision making through an assessment of both the current as well as planned future architectures [BMS09]. In this respect, the computation of quantified measures enables the comparison of different alternatives and the efficiency of investments [Iac+12].

Due to the diverse nature of the EA field, analysis techniques must be able to deal with a substantial amount of variability. One source of this variability is the large amount of competing standards. Canonical EAM frameworks - which describe the constituents and the layers of an enterprise architecture - include ArchiMate [HP12], TOGAF [The11], RM-ODP [ISO98] or the 4+1 view model of architecture [Kru95]. Furthermore, while the specifications of many of these standards follow the notion of modeling, they seldom provide reference implementations and often violate restrictions imposed by modeling frameworks such as MOF. Another complication arises from the fact that, in many cases, these languages require extensive customization to fulfill organization-specific needs.

Despite its obvious benefits, the concept of EA analysis is however scarcely explored in current industrial applications and research work as much effort is directed at improving the representational aspects of the EAM methodology [NBE12; Nie06]. It can be assumed that the complications of applying analysis in this domain contribute to this situation. In can be observed that most of the proposed analyses rely on either very generic techniques which do not take advantage of the rich information provided by modeling languages or employ proprietary methods which can only be applied under very specific circumstances. As a consequence, the adaption of the defined analyses or the specification of new analyses requires much effort. Techniques employed for EA analysis include, for example, XML [Boe+05b], SPARQL [SKR13], extended influence diagrams [Joh+07a], probabilistic relational models [Bus+11], p-OCL [Joh+13] or architecture theory diagrams [JNL07].

In the following paragraphs, we present a framework based on the DFA approach which enables the specification and execution of metamodel independent analyses. This framework employs a generic representation of architectural data for which arbitrary data-flow analyses can be defined, thus enabling a context-sensitive evaluation of organization-specific measures.

The application of the developed techniques will be demonstrated in the context of the *MIDWagen* example shown in Figure 10.22. This ArchiMate model is shipped with MID's *Innovator for Enterprise Architects* tool [MID14]. It describes the organizational structure of a car rental company, consisting of roles, business processes, applications and the underlying technical infrastructure.

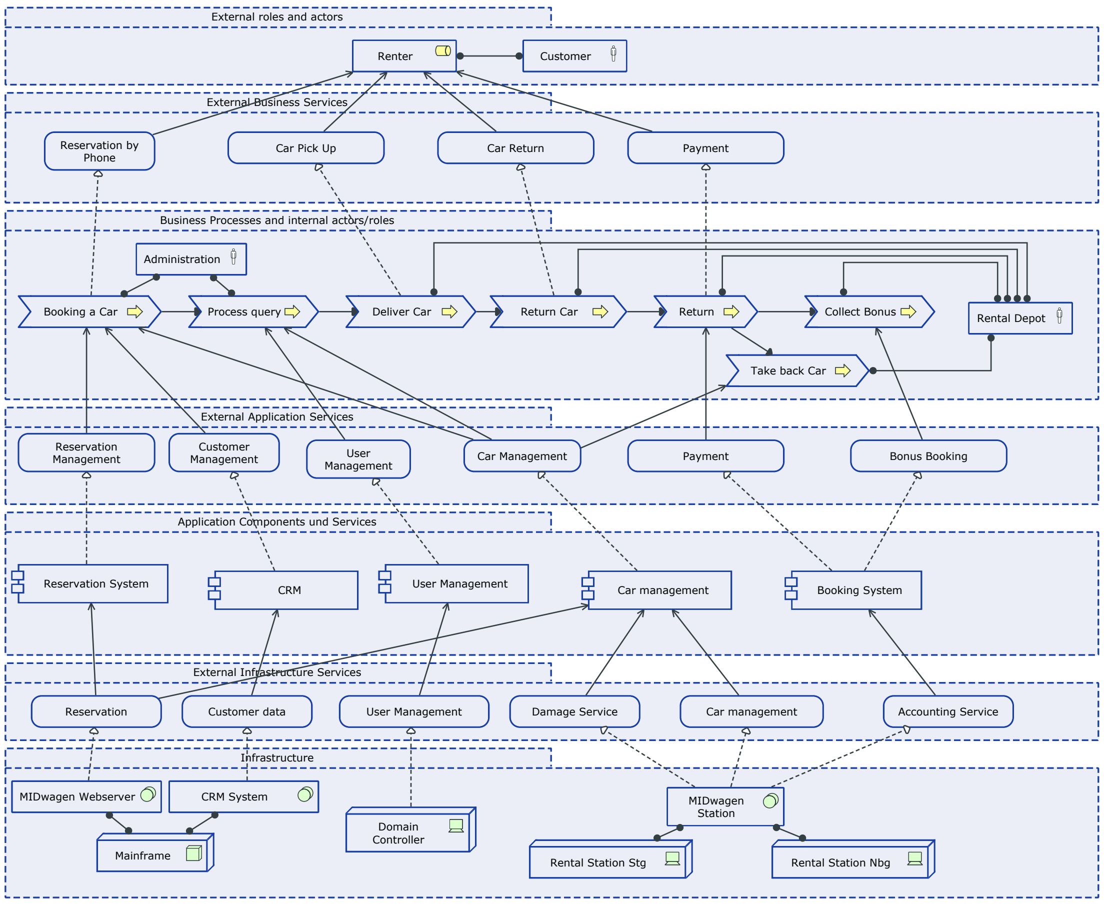


Figure 10.22.: *MIDWagen* example (ArchiMate representation) [MID14].

Generic Metamodel (GMM)

To derive meaningful information from a model, analyses must incorporate knowledge about the semantics of the language constructs. As mentioned above, this presents a problem in the EAM domain where a large number of competing standards and practices exist. To circumvent this complication, we introduce a generic metamodel (GMM) which acts as a universal interchange format and as common ground for the specification of analyses. Since the evaluation of a model element depends on its metamodel type, the GMM not only has to encode the actual model data but also the corresponding meta information, i.e. classes and associations of the EA language. To connect the model and the metamodel data, *«instanceof»* relationships must be established between both artifacts. As a consequence, each instance of the GMM conforms to a representation of the target model as well as the respective EAM language itself.

Figure 10.23 depicts the most important concepts of the generic metamodel[22] and groups them according to their artifact type. The root element EAModelContainer consists of an EAMetaModel, an EAModel and a Configuration, each of which in

[22]Additional types such as NamedElement have been omitted for reasons of clarity.

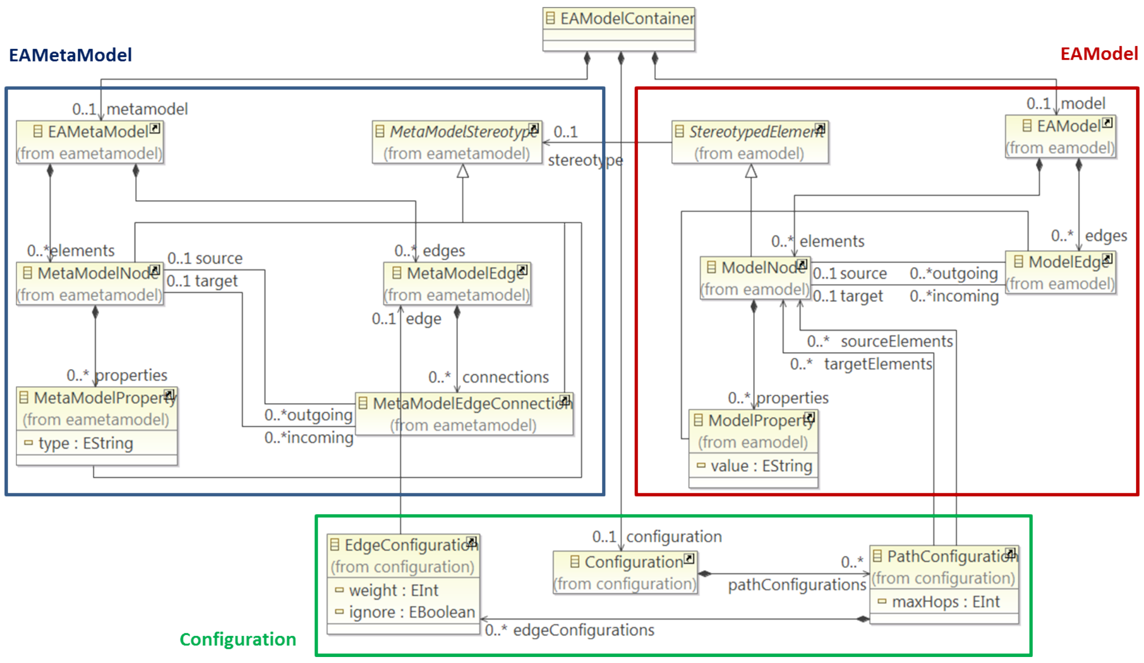


Figure 10.23.: Generic metamodel (GMM) for representing EAM (meta) data [LSB14a].

turn acts as a container for elements of the respective type. Generally speaking, this format describes a directed graph consisting of stereotyped elements[23]. For this purpose, the node, edge and property types of the EAMetaModel specialize the abstract MetaModelStereotype class while each "instance" of these elements in the EAModel is a StereotypedElement which links to its respective stereotype.

Mapping an existing EA language to the GMM format requires the translation of the classes to MetaModelNodes while their associated class attributes become MetaModelProperties. Edge types, on the other hand, require a mapping to two concepts: MetaModelEdge and MetaModelEdgeConnection. This distinction is necessary because the specifications of various EAM standards conflict with restrictions imposed by many common modeling frameworks. For example, implementations of ArchiMate often define multiple incoming or outgoing associations with the same name, a practice which is prohibited by MOF. Workarounds such as assigning a unique numeric index to each association have a significant downside: Since associations of the same type but with different indices are technically separate entities, this approach results in a loss of semantic information. The distinction made by the GMM solves this problem as a single MetaModelEdge type can be shared by multiple MetaModelEdgeConnections. As mentioned above, the EAModel represents an instance of the language encoded in the respective EAMetaModel. More specifically, it defines stereotyped elements in the form of ModelNodes along with ModelProperties and ModelEdges which establish connections between the nodes.

The Configuration artifact enables the encoding of analysis-specific data inside a GMM instance. With this setup, configurations can be directly tied to the ele-

[23]Nodes may additionally possess data fields.

ments of EAMetaModel and EAModel. Consequently, data-flow rules can access the configuration data and adjust their behavior accordingly.

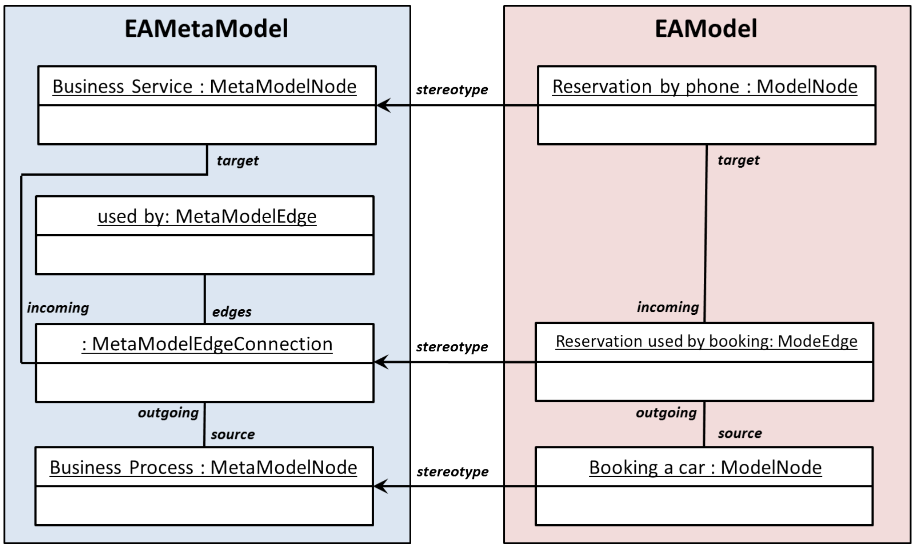

EAMetaModel
Business Service : MetaModelNode
target
used by: MetaModelEdge
incoming
edges
: MetaModelEdgeConnection
outgoing
source
Business Process : MetaModelNode
stereotype
stereotype
stereotype
EAModel
Reservation by phone : ModelNode
target
incoming
Reservation used by booking: ModeEdge
outgoing
source
Booking a car : ModelNode

Figure 10.24.: Example for the instantiation of the generic metamodel [LSB14a].

Figure 10.24 illustrates the principles behind the GMM using the *MIDWagen* example. The left hand side of the diagram represents EAMetaModel type definitions while the right hand side depicts the instance data.

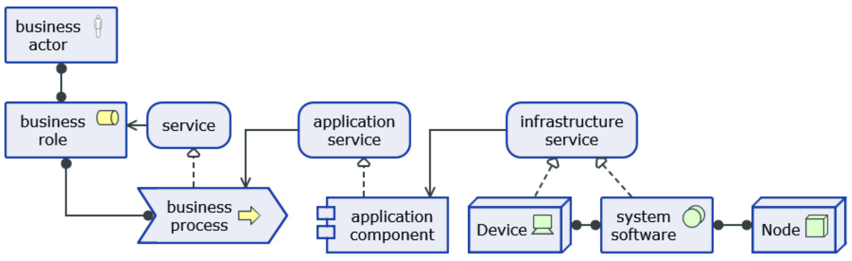


Figure 10.25.: Derived metamodel for the *MIDWagen* example [LSB14a].

In general, the easiest way to create a GMM instance for an existing EAM model consists of a traversal of the source model, creating corresponding GMM nodes, properties and edges on-the-fly. During this process, the respective meta information can be extracted as well, thereby generating an EAMetaModel which contains only the concepts which are actually used in the respective model. This principle is illustrated in Figure 10.25 which depicts the EAMetaModel of the *MIDWagen* model. Note that it only contains a subset of ArchiMate classes and relationships, namely the concepts which are actively used in the example.

10.2.2. Use Case: KPI Analysis

IT Continuity Coverage

To exemplify the computation of KPIs for EAM models, we chose to implement a measure for *"the coverage of IT continuity plans in respect to business-critical processes"* as defined by [Mat+12]. This indicator is specified as follows: *"Number of business-critical processes relying on business applications not covered by IT continuity plan divided by total number of business-critical processes"*.

The underlying (ArchiMate) language must therefore be extended with the following properties[24]:

- The boolean attribute IT continuity plan for application components
- The boolean attribute business critical for business processes

Algorithm 36 The attribution relevant_application_components

```
Attribution RELEVANT_APPLICATION_COMPONENTS
    attribute assignment relevantApplicationComponents : Set(ModelNode)
        initWith ∅
    extend ModelNode with
        occurrenceOf relevantApplicationComponents
            calculateWith modelNode_relevantApplicationComponents
```

```
Rule MODELNODE_RELEVANTAPPLICATIONCOMPONENTS(attrDef, context)
    for all (Path path : context[allpaths]) do
        if (not path.getTarget().type == "APPCOMPONENT") then
            continue;
        ignorePath ⇐ false
        for all (PathEntry pathEntry : path.getEntries() do
            if (pathEntry is ModelNode) then
                predecessor ⇐ pathEntry
            else if (not pathEntry.type == ("REALIZE" || "USEDBY") or
                not pathEntry.getSource() == predecessor)) then
                ignorePath ⇐ true
                break;
        if (ignorePath) then
            continue;
        relevantACs ⇐ relevantACs ∪ path.getTarget()
    return relevantACs
```

The KPI is computed by two data-flow attributes, relevantApplicationComponents and continuityCovered. While these attributes are not themselves propagated throughout the model, they nevertheless rely on context-sensitive information, namely the results of the allpaths analysis. This analysis is an adapted version

[24]This extension is supported by MID's Innovator tool.

of the template described in Section 9.2.4 and computes the paths starting at the local element to arbitrary target objects[25]. The first attribute represents the set of application components supporting a business process and can be calculated by evaluating the data-flow rule shown in shown in Algorithm 36 in the context of business processes. Note that this definition does not presume a specific path structure between business processes and application components. Instead, it relies on the value of allpaths to identify the relevant connection(s). Lines [3-4] ensure that only paths targeting an application component are regarded. For each alternative path leading to an application component, it is then checked whether the local business process actually relies on the target component. For this reason, the edge stereotypes must be either of type used by or realize [9]. Additionally, the connection has to be an incoming edge, i.e. opposed to the flow direction [10]. The path will be discarded if either of these two conditions is not met [13-14]. Otherwise, it is determined that the application component supports the business process and it is therefore added to the set of relevant components [15].

The second attribute, continuityCovered, evaluates to true for business processes if all of their supporting application components are covered by an IT continuity plan. Finally, the percentage of covered processes can be computed by dividing the amount of covered business critical processes by the number of all business critical processes in the model.

Performance Analysis

The authors of [JI09] propose the computation of performance metrics based on a top-down propagation of workloads and a bottom-up propagation of the utility. More specifically, these indicators encompass the *workload* λ, *response time* R, *processing time* T and *utilization* U. The following descriptions exemplify the calculation of the workload and the response time metrics.

The analysis relies on a set of properties which have to be incorporated into the EAM modeling language and configured by the enterprise architect:

- Weight n_e for any relation e
- Service time S_a for service a
- Capacity C_r for any resource r (actors, application components, devices and nodes)
- Arrival frequency f_a for business services and business processes

The workload λ_a for a node a in an architectural model is defined as:

$$\lambda_a = f_a + \sum_{i=1}^{d_a^+} n_{a,k_i} \lambda_{k_i}$$

[25]The Configuration concept of the GMM can be used to restrict the computation of model paths to sequences of a specific length(PathConfiguration→maxHops) and to ignore transitions of certain types (EdgeConfiguration→ignore).

where d_a^+ the out-degree and k_i a child of a.

Algorithm 37 Data-flow rules for the attribution performance_analysis

```
 1: Rule MODELNODE_WORKLOAD(attrDef, context)
 2:     result ⇐ node.arrivalFrequency
 3:     for all (ModelEdge outEdge : context.getOutgoing()) do
 4:         if (not outEdge.target.stereotype == context.stereotype) then
 5:             result ⇐ result + outEdge.weight * outEdge.target[workload]
 6:     return result
```

```
 1: Rule MODELNODE_RESPONSETIME(attrDef, context)
 2:     for all (ModelEdge inEdge : context.getIncoming()) do
 3:         if (inEdge.stereotype == "REALIZE") then
 4:             continue;
 5:         utilization ⇐ inEdge.source[utilization]
 6:         if (utilization >= 0) then
 7:             continue;
 8:         for all (ModelNode node : inEdge.source.getConnectedNodes()) do
 9:             if (not edge.stereotype == "ASSIGN") then
10:                 continue;
11:             utilization ⇐ node[utilization]
12:     return context[processingTime] / (1 - utilization)
```

This definition now has to be translated into a data-flow specification. Consequently, we define a data-flow attribute workload in the context of the class ModelNode. Instance results are computed by the data-flow rule modelNode_workload shown in Figure 10.26. Line [2] initializes the variable result with the value of the current node's arrivalFrequency. The loop in lines [3-5] then iterates over all outgoing edges. If the stereotype of the referenced object differs from the type of the context object, the result is updated accordingly. This check ensures that connections between elements of the same type, for example between two business processes, are ignored [4]. Then, the value of the edge's weight is multiplied by the workload data-flow result computed at the edge's target node [5].

The response time R_a for a node a in an architectural model is defined as:

$$R_a = \frac{T_a}{1 - U_{r_a}}$$

where r_a denotes the realizing resource of a.

The realizing resource r_a can be either directly connected to the service or indirectly via a behavior element. Therefore, the rule modelNode_responseTime first iterates over all incoming edges [2] and ensures that they are of the type realize [3]. If available, the utilization value of the respective source element will be used [5]. Otherwise, a nested loop processes all indirectly connected elements and tries to request their utilization value [8-11]. Utilization and processing time can also be specified as data-flow equations according to the formulas in [JI09].

Application Usage

The authors of [NBE12] present a framework for multi-attribute information system analysis which includes an evaluation of *application usage*. This measure addresses voluntary application usage, i.e. it determines why users employ a specific application. The result for the usage attribute at application components can be derived through the following linear regression model:

$$usage = \alpha + \beta_1 * TTF_1 + ... + \beta_n * TTG_n + \beta_{n+1} * PU + \beta_{n+1} * PEoU$$

Again, the analysis relies on a number of properties which must be specified by the enterprise architect:

- α, β_1, ...: Constants determined by processing empirical survey of data of application usage
- PU: Perceived Usefulness
- $PEoU$: Perceived Ease of Use
- TTF: Task Technology Fit (derived from the class attributes Task Fulfillment and Functionality)

This use case depends on the properties Task Fulfillment for business processes, Functionality for application services and application functions, PU and PEoU for actor-component relationships, regression coefficients TTF, PU, PEoU and finally a Domain Constant for application components. For this use case, the modeling language also had to be extended with the relationship used by which connects application components and actors.

For the analysis of the application usage, we can define the following data-flow attributes: The attributes weightedTTF for used by are computed for edges linking application services to business processes while weightedTAM is annotated at application components. The weightedTTF is calculated for each used by edge connecting an application service to a business process. The values of these attributes are propagated to the corresponding application components via the application functions of those services. The weightedTAM (weighted PU + PEoU) is calculated for each application component by iterating over all outgoing edges to an actor. To get the final usage result, each component has to add both values to the respective domain constant.

10.2.3. Use Case: Impact Analysis

Impact analysis simulates the effects of architectural changes on an application landscape or an enterprise architecture. It is therefore an important tool which enables enterprise analysts to assess risks in the current architecture, for example which business operations may be affected if a specific server goes offline [Boe+05a].

Because the effect of an architectural change on a neighboring element (*direct impact*) may in turn propagate to its neighbors (*indirect impact*), even a small change in a single element can result in non-trivial consequences. Determining whether an element is affected and how its status changes therefore requires a contextual analysis of the (transitive) connections to other elements. Furthermore, different relationship types may have different semantics with respect to the propagation of changes. For example, to evaluate the impact of a server failure on a company's business processes, the analysis must first examine which applications rely on this server before computing the effect on the processes. This principle is also mentioned by [Boh02] who notes that determining the effects of a change requires an iterative and discovery-based approach.

Since the computation of (in)direct change impacts (*n-level impacts*) requires the consideration of transitive paths, information must be propagated in a way that ensures that all necessary information is regarded while at the same time excluding irrelevant connections. However, due to the fact that EA models only provide scarce information about the technical details of the modeled systems, an approximation of change effects tends to overestimate the result by generating false-positives. For this reason, the presented change propagation approaches compute results for a best case as well as a worst case scenario. The result for the best case represents the minimal amount of affected elements while the worst case reflects all potential impacts. To assess the actual risks, which can be expected to lie somewhere in between, the respective result sets must be interpreted by domain experts.

The presented analyses differentiate between the change types *extend*, *modify* and *delete* as proposed by [Boe+05a]:

- An *extension* (*ext*) represents a change which preserves the initial functionality or structure.
- While a *modification* (*mod*) also affects the functionality/structure, in this case, it cannot be guaranteed that the components will still be available or that their behavior remains unchanged.
- *deletion* (*del*) indicates removal of an element from the enterprise architecture.
- Finally, *no change* (*no*) represents changes which have no effect on the target.

The prioritization between these types is defined as follows: *delete* > *modifies* > *extends* > *no change*. If required, it is possible to extend this definition with organization-specific change types.

Change Propagation based on Relationship Classes

One possibility to assess the impact of changes consists of a classification of relationship types according to their semantics. A literature review which included the

Core Concepts Model (CC) of ArchiMate [HP12] and the DM2 Conceptual Data Model of DoDAF [US 10] resulted in five classes of relevant EA relationship types:

- *locate* denotes allocation to a location or an organization unit.
- *provide* represents provisioning of functionality, information or behavior.
- *consume* denotes consumption of an element.
- *structural dependency* relationships define the structural aspects of entities.
- *behavioral dependency* connections describe dependencies between behavioral elements which are neither of the type *provide* nor *consume*.

Mapping relationship types to these classes enables the analysis to choose a suitable propagation rule for the respective type. If a relationship belongs to more than one category, the worst case analysis applies the strongest rule while the best case analysis relies on the weakest one. It should be noted that relationship-to-class mappings are always based on a specific interpretation of the relationships' semantics and can be adapted if necessary.

Propagation rules can be specified using the following syntax:

$$A.\{X_1, X_2, \dots X_n\} \rightarrow BC : B.Y,\ WC : B.Z$$

with objects A and B representing the source and the target of a relationship and $X_1, X_2, \dots X_n, Y, Z \in \{ext, mod, del, no\}$. The statement on the left hand side encodes the prerequesite while the right hand side denotes the effect on the respective source or target in the best case (BC) and worst case (WC) scenario.

Table 10.7 lists a set of propagation rules for the five classes. As an example, we assume that the relationship type which indicates that an application component A is hosted by an organization unit B is mapped to the *locate* class. The rule $A.\{del, mod, ext\} \rightarrow B.\{no\}$ indicates that a change to the application component has no effect on the organization unit. However, if the organization unit is deleted, the application component must be assigned to a new host (best case) or has to be deleted as well (worst case). Finally, in the worst case scenario, a modification or extension of the organization unit will also require a modification of the component.

Change Propagation based on Effect Classes

An alternative classification methodology relies on the severity of an effect's impact. Propagation rules for the three categories *strong*, *weak* and *no effect* must be specified for both directions. The notation $X - Y$ indicates that a change in the source has a effect of type X on the target and vice versa. This leads to six effect classes: *strong-strong*, *strong-weak*, *strong-no effect*, *weak-weak*, *weak-no effect* and *no effect-no effect*.

Table 10.8 defines propagation semantics for the three effect types. If A has a *strong* effect on B, a deletion of A results in an extension (best case) or a deletion

class	rule

`locate`	A.{*del,mod,ext*} → B.{*no*}
	B.{*del*} → BC: A.{*ext*}, WC: A.{*del*}
	B.{*ext,mod*} → BC: A.{*no*}, WC: A.{*mod*}

`provide`	A.{*del*} → BC: B.{*ext*}, WC: B.{*del*}
	A.{*mod*} → BC: B.{*no*}, WC: B.{*mod*}
	A.{*ext*} → BC: B.{*no*}, WC: B.{*ext*}
	B.{*del,mod,ext*} → A.{*no*}

`consume`	A.{*del,mod,ext*} → B.{*no*}
	B.{*del,mod*} → BC: A.{*ext*}, WC: A.{*mod*}
	B.{*ext*} → A.{*no*}

`structural dependency`	A.{*del*} → BC: B.{*mod*}, WC: B.{*del*}
	A.{*mod,ext*} → B.{*no*}
	B.{*del,mod*} → BC: A.{*no*}, WC: A.{*mod*}
	B.{*ext*} → BC A.{*no*}, WC: A.{*ext*}

`behavioral dependency`	A.{*del,mod,ext*} → B.{*no*}
	B.{*del,mod,ext*} → A.{*no*}

Table 10.7.: Impact rules for the relationship classes (adapted from [LSB14b]).

effect	rule

`strong`	A.{*del*} → BC: B.{*ext*}, WC: B.{*del*}
	A.{*mod*} → B.{*mod*}
	A.{*ext*} → B.{*ext*}

`weak`	A.{*del*} → BC: B.{*no*}, WC: B.{*mod*}
	A.{*mod*} → BC: B.{*ext*}, WC: B.{*mod*}
	A.{*ext*} → BC: B.{*no*}, WC: B.{*ext*}

`no effect`	A.{*del,mod,ext*} → B.{*no*}

Table 10.8.: Impact rules for the effect classes (adapted from [LSB14b]).

(worst case) of B while both a modification and an extension of A directly transfers to B. As a concrete example, we assume that the *realization* of a service has been mapped to the `strong-weak` class, i.e. the component has a `strong` impact on the service while the effect in the opposite direction is classified as `weak`. As a result, the deletion of the service has either no effect on the component (best case) or it requires a modification (worst case).

Data-flow based Change Propagation

The propagation of changes, whether based on a categorization of relationships or effects, can be implemented as a data-flow analysis. Just like in the KPI use case, the analysis can be specified for the generic metamodel to ensure that its application does not depend on the specifics of an EAM framework.

Results are computed by a data-flow attribute named **status** which is initialized with the change configuration. For this purpose, attribute instances at the affected elements are set to *ext*, *mod* or *del*, depending on the change that should be simulated, while all other values are initialized with *no change*. To avoid hard coding the change configuration in the analysis specification or the EA model, it is possible to supply this information via the GMM instance's Configuration and process it in the DFA's initialization rules.

Algorithm 38 Data-flow rule for the attribution impact_analysis

```
 1: Rule modelNode_changeStatus_bestCase(attrDef, context)
 2:     currentStatus ⇐ context[status]
 3:     for all (ModelEdge edge : context.getIncomingEdges()) do
 4:         sourceStatus ⇐ edge.source[status]
 5:         if (edge == StrongEffectTarget) then
 6:             if (sourceStatus == (DEL||EXT)) then
 7:                 return computeStatus(currentStatus, EXT)
 8:             else if (sourceStatus == (MOD)) then
 9:                 return computeStatus(currentStatus, MOD)
10:         else if (edge == WeakEffectTarget) then
11:             if (sourceStatus == (DEL||EXT)) then
12:                 return computeStatus(currentStatus, NO)
13:             else if (sourceStatus == (MOD)) then
14:                 return computeStatus(currentStatus, EXT)
15:         else if (edge == NoEffectTarget) then
16:             if (sourceStatus == (DEL||MOD||EXT)) then
17:                 return computeStatus(currentStatus, NO)
18:     for all (ModelEdge edge : context.getOutgoingEdges()) do
19:         targetStatus ⇐ edge.target[status]
20:         ...
```

Algorithm 38 demonstrates how propagation rules can be implemented for the effect classification method. The data-flow rule modelNode_changeStatus_bestCase iterates over the incoming [3-17] and outgoing [18-20] edges to determine the appropriate propagation rule for the current context. For this purpose, the conditions in lines [5,10,15] evaluate the effect type associated with the incoming edge. Subsequently, the new status is computed based on the current result and the status of the edge's source node. The prioritization relationships between the different change types are implemented in the method computeStatus() which ensures that a weak change will not overwrite a stronger one. Outgoing edges are computed accordingly.

As with the initial change configuration, it is possible to encode the mappings for relationship and effect classes in the GMM's Configuration. This way, the analysis can be defined in a generic manner while organization-specific properties are supplied by the respective model transformation which generates the GMM instance.

10.2.4. Tooling Integration

The market offers multiple tools which support the development of EA models. Examples include Innovator [MID14], PlanningIT[26] and iteraplan[27]. Many of these applications provide a plugin interface which can be used to extend the existing modeling facilities with custom features.

In the original publications on which this case study is based, MID's Innovator platform has been chosen to demonstrate the viability and applicability of the proposed methods. However, since this tool is not based on the Eclipse RCP framework and does not provide support for Java-based plugins, it was not possible to directly integrate MAF's core libraries which are necessary to execute the specified analyses. The development of a native reimplementation of the analysis facilities did not seem feasible due to the large effort which would be required to carry out this task.

Instead, we opted for a solution which did not necessitate a duplication or modification of any existing components or functionality. For this purpose, the Model Analysis Framework was wrapped in a server container[28] which enables external applications to access the analysis facilities using network interfaces. Furthermore, a Xtext-based DSL was developed for the generic metamodel which allows GMM instances to be serialized and transmitted via TCP/IP sockets. An Innovator plugin translates the selected model data into this textual notation. Innovator-specific relationship mappings and user settings for the respective evaluation run are encoded in the GMM's Configuration object. The serialized model can then be sent to the MAF analysis server which deserializes the model, executes the analyses and transmits the results back to the plugin. In the final step, the Innovator plugin generates and displays a suitable visualization for the computed results.

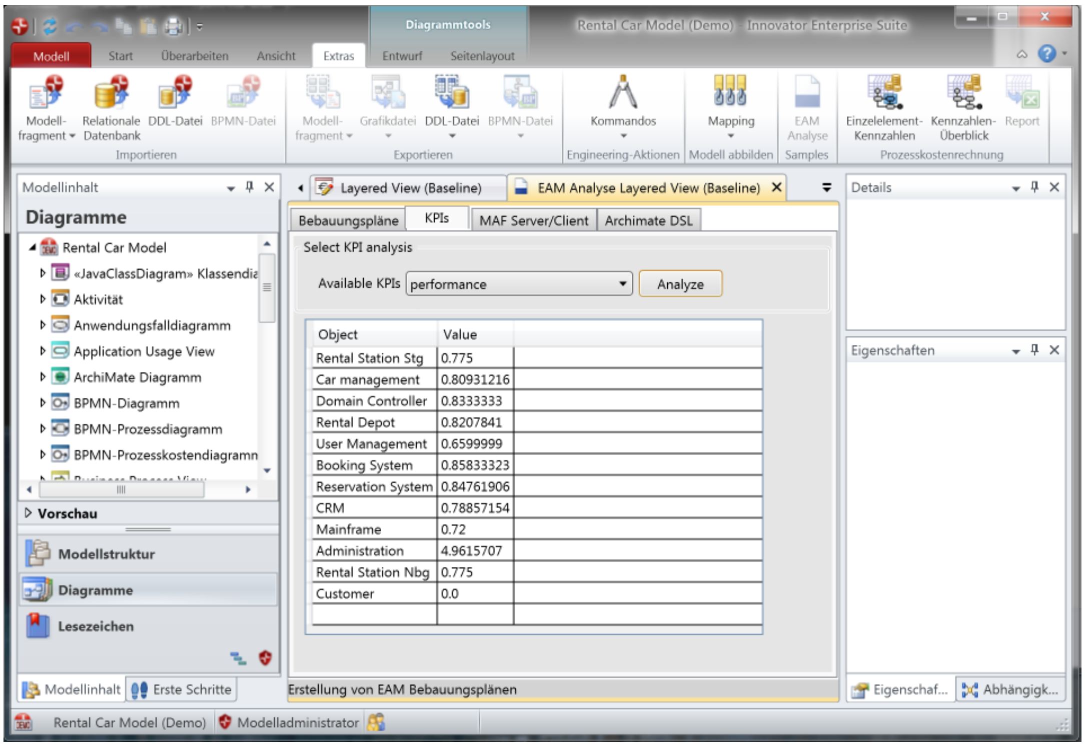
The screenshot shown in Figure 10.26 depicts the results of the *performance* analysis computed for the *MIDWagen* model and displayed by the developed Innovator plugin.

10.2.5. Evaluation of the Use Cases

The viability of the proposed analysis framework for the EAM domain has been demonstrated in the context of the *MIDWagen* example and a prototypic implementation for the Innovator tool suite. Moreover, it has been shown that the flow-based approach to model analysis provides a powerful method for the specification and evaluation of key performance indicators in this domain. To test the generic applicability of this technique, several existing methods described in canonical literature and research publications have been reimplemented.

Each of the presented use cases required specific adaptions of the underlying modeling language. This reflects the general situation in the EAM domain where many competing standards exist which are often customized to fulfill organization-specific needs. Furthermore, some of these standards may conflict with the restrictions

[26] http://www.alfabet.com
[27] http://www.iteraplan.de
[28] This has been discussed in Section 8.4.



Select KPI analysis

Available KPIs: performance — Analyze

Object	Value
Rental Station Stg	0.775
Car management	0.80931216
Domain Controller	0.8333333
Rental Depot	0.8207841
User Management	0.6599999
Booking System	0.85833323
Reservation System	0.84761906
CRM	0.78857154
Mainframe	0.72
Administration	4.9615707
Rental Station Nbg	0.775
Customer	0.0

Figure 10.26.: Screenshot of Innovator showing *performance* metrics computed for the *MIDWagen* example.

imposed by modeling frameworks. These problems have been addressed by the development of a unified representation of EA model and metamodel data. This method also solved another problem: Structural changes of the underlying modeling language also affect the specification of analyses. For example, the metamodel shown in Figure 10.25 allows business roles and business processes to be connected either directly or indirectly via intermediary services. Customizations of the modeling language may limit these options or introduce additional alternatives. This means that even the analyses themselves must cope with a certain amount of variability. The propagation of data-flow values along model edges presents a solution to this problem as it abstracts from the concrete structure by making information available along transitive paths.

This principle is, for example, applied in the computation of the *IT continuity* metric where the identification of relevant paths happens independently from the concrete structure of the (meta)model and the language-specific mappings which can be supplied without changing the analysis itself. The proposed framework therefore represents an improvement over the methods used in the respective original publications: The flow-based implementation of the performance analysis is not limited to ArchiMate models, does not require a normalization step and does not presume a 1-to-1 mapping of a behavior element to a service/resource. In contrast to the implementation described in [NBE12], our technique does not require an extension of the model with intermediate helper entities to support the aggregation of values.

Similar advantages can also be identified in the impact analysis use case. While

[Boe+05a] presents rules for computing the change impact on an architectural element, the definitions are given in an informal and textual manner and no technical realization exists. A similar situation exists with [Kum+08] which does not specify a concrete mechanism for the implementation of the change propagation.

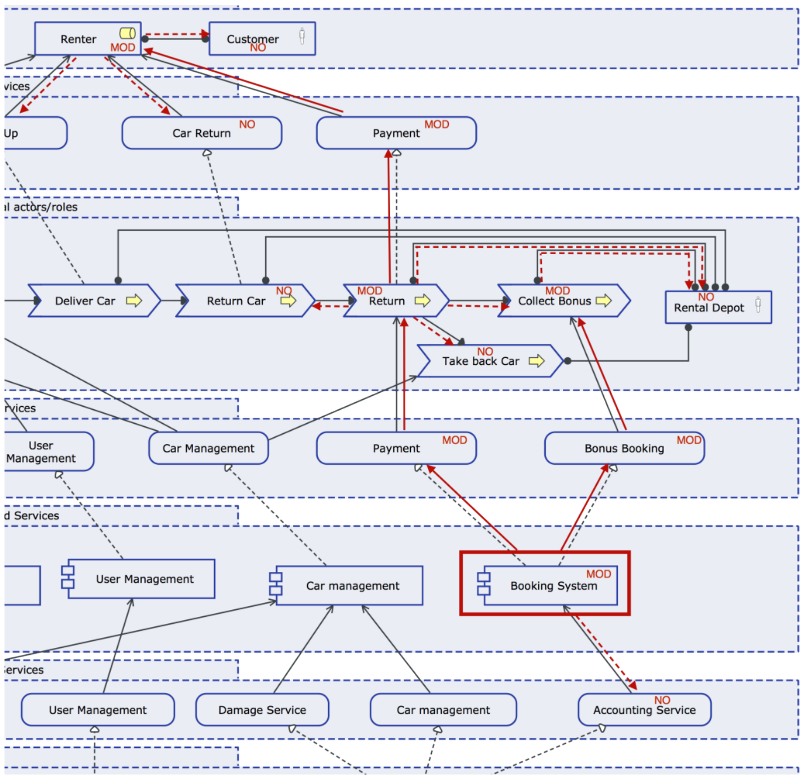


Figure 10.27.: Worst case change propagation path for the *MIDWagen* model [LSB14b].

The usefulness of the proposed methods for impact analysis can be demonstrated in the context of the *MidWagen* example. Figure 10.27 depicts the impact of a modification of the Booking System service. The red arcs indicate the worst case change propagation paths. Solid lines represent paths along which a change is forwarded while dashed lines denote relationships which do not result in change propagation. The final impact set consists of the elements {Payment AS: *modified*, Bonus Booking: *modified*, Return: *modified*, Collect Bonus: *modified*, Payment BS: *modified*, Renter: *modified*}. The effect classification method generates the following result set: {Payment AS *modified*, Return *modified*, Payment BS *modified*, Renter *modified*}. In both instances, the best case scenario yields no required modifications.

It can be argued that the results for both the best and the worst case represent realistic approximations since the final assessment must consider the severity of the modification. In the best case scenario (e.g. performance issues), an upgrade of the Booking System service will not affect its functionality. In the worst case, i.e. a substantial modification of the service, the change may cause other components to stop working, thereby affecting the business layer. Although the *MIDWagen* model is not a real world example, the level of detail and the extensibility of the underlying

EA language enables a thorough evaluation of the viability and the robustness of the proposed methods. In essence, the computed impact sets allow an estimation of the full effects of planned changes or unexpected incidents on the IT landscape of a company and therefore can be of great value to an enterprise. Since the developed analysis distinguishes between different semantic relationship classes it can be easily adapted to the conventions in different organizations by mapping the relationship types in the respective target domain to the proposed categories. Furthermore, it is possible to extend the analysis with individual impact propagation rules.

In conclusion, the combination of the generic metamodel, dynamically configurable flow analyses and classifications of relationships and effects ensures that this technique can be used to implement various analysis scenarios and supports the diverse EA conventions found in different organizations.

10.2.6. Summary and Discussion

In this case study, we presented two analysis methodologies for the EAM domain. The described use cases focus on the computation of KPIs [LSB14a] and the assessment of the impacts of architectural changes [LSB14b]. Both techniques rely on a combination of a generic representation of EA data (GMM) and flow-based model analyses to implement a robust and powerful analysis framework.

By abstracting from the concrete representation of a specific EAM methodology, the generic metamodel provides support for different standards and practices as well as for organization-specific adaptions. The GMM is therefore an important prerequisite for dealing with the variability encountered in this domain and provides a common ground for analysis specifications. This principle also applies to the analyses themselves which must be defined in a generic and configurable manner as the data-flow rules cannot presume the existence of a specific metamodel structure. This is addressed by the GMM's Configuration element which can encode language-specific mappings as well as user settings for a concrete evaluation run. This enables the data-flow rules to adjust their behavior dynamically. Some analyses rely on the identification of paths which start and end at elements conforming to a specific stereotype. Since the underlying modeling language may be subject to changes, it would not be possible to rely on fixed navigation statements to address distant objects. The allpaths analysis (cf. Section 9.2.4) provides a solution to this problem as it computes these paths dynamically.

We will now summarize our findings:

Implementation effort

The presented use cases introduced some complications such as the variability of meta data and the integration of analysis capabilities into a proprietary third-party tool. While some of these challenges are specific to the EAM domain, others may occur in other areas as well. As a consequence, it was necessary to develop a suitable overall methodology which addresses these problems by providing a generic approach to the specification and configuration of analyses. Using the capabilities offered by the technological space of modeling, it was possible to address these issues with little effort by implementing a model-based representation of EA meta and model data. Furthermore, this specification was supplemented with a textual DSL for the (de)serialization of model artifacts which enabled the realization of an analysis frontend for the chosen tooling platform.

Classification
The computation of KPIs represents a typical application scenario for model analyses in combination with a derivation of control-flow properties (cf. Section 9.2.4). Impact analysis also falls in the same category as it examines the structural properties of models. However, in this case, the analysis is stimulated with external information required for specific evaluation runs. More specifically, it relies on the specification of an initial change configuration. The overall methodology demonstrates that the flow-based analysis approach can be applied to use cases with specific requirements.

Reuse
Since the computation of KPIs and change impacts is a highly domain-specific application scenario, the described analyses represent mostly new methods which - with the notable exception of the allpaths analysis - are not part of the standard library. In fact, the implementation of the use cases required the development of several new methodologies. These include the generic metamodel, dynamically configurable analyses and the integration of analysis capabilities in a non-Eclipse platform. It should be noted that these methods may be beneficial in other domains as well and should therefore be elaborated in future research work.

Usability
It has been demonstrated that the use cases can be implemented as plugins for the Innovator tool suite. This allows enterprise architects to configure and invoke the respective computations from the modeling environment. Furthermore, the visualization of the results also takes place inside this tooling, thereby providing a unified interface for executing the analyses and reviewing the results. The *MIDWagen* example showed that the proposed methods can be of great benefit when dealing with the challenge of assessing the current or future state of an IT infrastructure.

Performance
Although the original publications do not provide an assessment of the performance aspects of the proposed analyses, it can be assumed that they can be executed in a satisfactory time frame. While both the KPIs and the change impacts rely on the propagation principle, they do not compute fixed-point results. Because each rule must therefore be executed only once to calculate the final value, the amount of necessary invocations is proportional to the size of the model. A potential bottleneck can be found in the allpaths analysis which tends to generate large result sets as the addition of new elements drastically increases the amount of possible routes between objects. However, this

problem can be addressed by limiting the maximal length of the derived paths or by ignoring irrelevant relationship types.

In [LSB14a] and [LSB14b], we discussed several possible enhancements to extend the capabilities and improve the usefulness of the EAM analysis frameworks.

It is stated that propagation rules could be employed to quantify changes, e.g. in terms of costs, to augment the computation of change impacts with an estimation of the expected costs or savings. Data-flow attributes could, for example, be used to aggregate the maintenance costs of all deleted application and infrastructure components and their corresponding services to compute potential savings on IT maintenance.

Further proposals refer to the implementation of a failure impact analysis which examines the availability of architecture elements or the use of probability distributions to improve the assessment of the best and worst case scenarios. The latter proposition would require to include change probabilities in the propagation rules and to compute a separate impact for each change type. Rules could be defined in the following fashion:

$$P(A.\{del\}) = X \rightarrow P(B.\{del\}) = 0.8 \times X$$

This can be interpreted as follows: If the probability that A is deleted is X, then the probability that B has to be deleted is $0.8 \times X$ or, in other words, if A is deleted then in 80% of the cases B will be deleted as well.

Another suggestion discusses an extension of the GMM with support for specialized relationship types such as generalization or containment. By encoding this information in the generic metamodel, analyses would be able to directly process these types without the need for additional mappings. It is also mentioned that it should be explored how the visualization of the metrics and change impacts could be improved. It can be assumed that a graphical representation in the diagram editor would enable a quick and intuitive assessment of the IT infrastructure's state.

10.3. Case Study: SE-DSNL

In this case study, we will examine the application of data-flow analysis in the context of Natural Language Processing (NLP). The approach presented in this section employs ontology-driven methods to enhance traditional NLP techniques with semantic annotations to improve recognition and classification of natural language text. Detailed in [Fis13], it uses a concept termed *Semantically Enhanced Domain-Specific Natural Language (SE-DSNL)*[29] and heavily employs model-driven techniques, namely a MOF-based domain-specific language, to support knowledge engineers in their task of encoding information about the semantics of concepts and relationships in the target domain. The approach connects natural language

[29]The approach evolved from the research project BRM 3.0 (`http://www.informatik.uni-augsburg.de/de/lehrstuehle/swt/vs/projekte/semtech/brm30/`) and was formerly named BRM-S (Semantically enabled Business Requirements Management).

constructs to ontological specifications and uses this information to provide better support for NLP-specific tasks.

As is the case with most domain-specific languages, SE-DSNL language expressions must conform to a set of restrictions which constitute the static semantics of the language. It is vital that SE-DSNL models respect these well-formedness rules so that they can be interpreted correctly by the corresponding NLP algorithms. The theoretical approach described in [Fis13] is complemented with an integrated, EMF/GMF-based tooling environment [Loh13]. Consequently, this approach is a perfect candidate for a practical evaluation of the DFA method as we can study its application in the context of what constitutes a common usage scenario: The starting point is a model-based DSL with a set of informally defined well-formedness restrictions. In fact, a complete MAF-based validation framework for the SE-DNSL suite has been developed under the guidance of the author of the SE-DSNL approach.

In the following sections, we will shortly introduce the motivation and the principles behind SE-DSNL and the syntactic and semantic restrictions that must be met by valid language expressions. This is followed by several examples of analyses which have been implemented in the aforementioned validation framework. Finally, the technical integration is described alongside a statistical evaluation followed by a discussion of the case study.

10.3.1. Introduction and Motivation

Most researchers would agree that, with the exception of a few very specific application scenarios, the interpretation of natural language expressions (NLP) is a problem that has not been - and probably will not be - solved in a satisfactory way for quite some time. Many factors contribute to the difficulties encountered in this area including the relevance of contextual information or ambiguities in the wording.

In many cases, contemporary NLP techniques parse natural language expressions and try to derive a mapping between the syntactical representation and semantic interpretations. These mappings can then be further evaluated to achieve certain goals (e.g. to implement a question/answer system). The author of [Fis13] notes that most state-of-the-art NLP systems implement this principle using a pipeline-driven approach. In essence, these analyses perform a syntactic recognition and composition of part of speech (POS) fragments in an effort to detect the grammatical structure of sentences and thus the relations between the syntactic constituents. It is stated that this approach can be problematic as smaller errors in earlier stages of the pipeline may have a more severe impact in later stages which is major reason for imprecise or incorrect results.

As a remedy for this situation, the proposed solution employs the principles of Cognitive Linguistics [CC04] to improve the recognition process. The research area of Cognitive Linguistics operates on the assumption that different linguistic levels must be examined in parallel in an effort to mimic the speech recognition processes in the human brain. For this purpose, the SE-DSNL approach uses so-called *construction rules* which associate syntactic constructs with semantic representations. This combines the processes of syntactic recognition and semantic analysis by assembling semantic interpretations for given texts from the syntactic fragments.

SE-DSNL makes heavy use of the ontology concept to model the syntactic and semantic properties of language constructs [FB10; FB11]. The area of ontology research has become more prominent in recent years, especially in the context of the Web Ontology Language (OWL) [OWL09] which attempts to make information which is available on the internet usable by computers. An ontology can be viewed as a knowledge graph that connects concepts and instances through qualified associations. Consequently, this formalism bears some resemblance to the technological space of modeling. Ontologies are however based on rigid logical specifications[30] to enable logical reasoning on the modeled knowledge base.

The authors emphasize that ontologies expressed in standards such as OWL or RDF(S) are not very useful for NLP. Although these methods support the encoding of domain-specific semantic relationships, they do not provide sufficient means to enrich the ontological definitions with the complex linguistic information that is necessary to "reverse engineer" semantic information from the syntactical structure of a language expression. To circumvent these limitations, a MOF-based ontology DSL is proposed which encapsulates the relevant features of ontologies while additionally supporting the annotation of linguistic information.

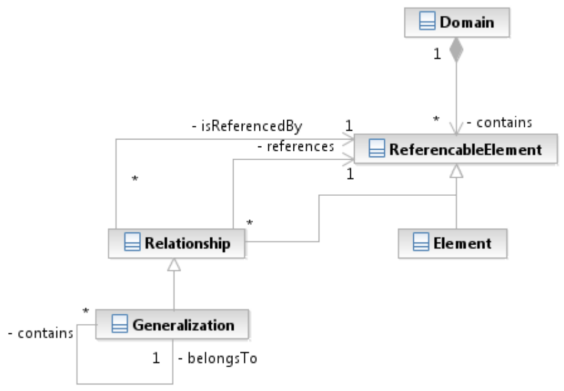


Figure 10.28.: Reference Scope of the SE-DSNL metamodel [FB10].

The SE-DSNL metamodel can be roughly divided into three interconnected segments representing different aspects of the same concepts: The Syntactic Scope, the Semantic Scope and the Construction Scope. The Syntactic Scope formalizes the lexical representations of words including, for example, different inflections of the same word. The Semantic Scope, on the other hand, categorizes the relevant elements of the respective domain and describes their relationships. It therefore conforms to an ontological representation of the target domain. In this case study, we will mainly focus on the Construction Scope and examine several analyses in this context. The Construction Scope combines syntactical and semantic specifications in the notion of Cognitive Linguistics, i.e. it interweaves the syntactic construction rules with information from the Semantic Scope.

[30]Ontologies are often based on *description logics* which represent a subset of first-order predicate logic. For more information see http://dl.kr.org/.

The three scopes are all based on a common infrastructure called Reference Scope. Shown in Figure 10.28, this specification defines the underlying concepts and relationships shared by the three main scopes. Although there are differences both in structure and in semantics when compared to the well-known MOF/UML, in the context of this case study we can assume that these concepts have a similar meaning. For example, Elements correspond to MOF classes while Relationships denote references between these types.

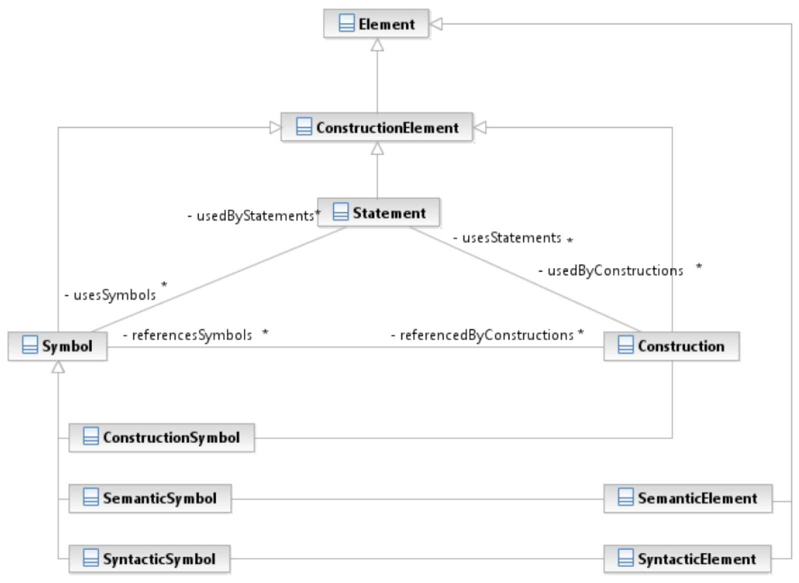


Figure 10.29.: Construction Scope of the SE-DSNL metamodel [FB10].

Figure 10.29 shows an excerpt of the SE-DSNL metamodel depicting the elements of the Construction Scope and their relations to classes from the Semantic Scope. The most important concept here is the Construction class which combines information from the syntactic and the semantic domains: *"Inheriting from ConstructionElement, the central element of the ConstructionScope, it consists of a set of Symbols as well as Statements. A Symbol can be used to reference any kind of information, be it SemanticElements (SemanticSymbol), SyntacticElements (SyntacticSymbol) or other Constructions (ConstructionSymbol)"* [Fis13].

10.3.2. Static Semantics and Guidelines

In order to ensure that a given SE-DSNL ontology model is sound, it must adhere to a set of restrictions (static semantics) which relate to the structural composition of the model's elements. [Fis13] lists a set of modeling guidelines which detail these requirements. For the purpose of this case study, a subset of guidelines which apply to both the Semantic Scope and the Construction Scope has been selected. A condensed version of these descriptions will be presented in the following paragraphs.

Predicates

Let S be the set of all elements contained within the Semantic Scope of a metamodel instance. Then $S_e \subseteq S$ is the set of all SemanticElements in S. $S_a \subseteq S$ is the set of all Associations in the Semantic Scope.

$gen(a, b)$ validates for $a, b \in S_a$ if b is a Generalization of a. The Generalization relationship is transitive

$equal(a, b)$ $:= a == b$, i.e. it validates if two elements a and b are the same

$hasType(a)$ where $a \in S_a$, validates if a has a type, i.e. if $a.type \in S_e$

$isType(a, s)$ validates for $a \in S_a, s \in S_e$ if s is the type of the Association a

$isSource(a, s)$ validates for $a \in S_a, s \in S_e$ if s is the source of the Association a

$isTarget(a, s)$ validates for $a \in S_a, s \in S_e$ if s is the target of the Association a

Restrictions on the Semantic Scope

S1 $\exists e \in S_e \; \forall m \in S_e \; gen(m, e)$, i.e. all SemanticElements must be part of a single Generalization hierarchy

S2 Multiple inheritance is allowed

S3 $\forall e, f \in S_e \; \neg(gen(e, f) \wedge gen(f, e)) \vee equal(e, f)$: Generalization links are not allowed to form circles.

S4 $\forall a \in S_a \; hasType(a)$: Each Association must contain a type

S5 The type of an Association $a \in S_a$ must be part of a different Generalization branch than the source and target elements of the Association: Let $o \in S_e$ be the root element of the ontology, $e_s \in S_e$ the source of an Association, $e_y \in S_e$ be the type and $e_t \in S_e$ the target of the Association. Then $\forall m_s, m_y, m_t \in S_e \; isSource(a, e_s) \wedge isTarget(a, e_t) \wedge isType(a, e_y) \wedge gen(e_s, o) \wedge gen(e_y, o) \wedge gen(e_t, o) \wedge gen(m_s, o) \wedge gen(m_y, o) \wedge gen(m_t, o) \wedge gen(e_s, m_s) \wedge gen(e_y, m_y) \wedge gen(e_t, m_t) \wedge \neg equal(m_y, m_t) \wedge \neg equal(m_s, m_y)$

S6 The information within the Semantic Scope must always be modeled either in an active or a passive representation

Restrictions on the Construction Scope

C1 Constructions must be organized in a single Generalization hierarchy

C2 Multiple inheritance is **not** allowed

C3 $\forall e, f \in S_e \; \neg(gen(e, f) \wedge gen(f, e)) \vee equal(e, f)$: Generalization links are not allowed to form circles

C4 Atomic Constructions always reference exactly two symbols, one SyntacticSymbol and one SemanticSymbol

C5 The desired level of detail for construction rules results in a trade-off between *precision* and *recall*[31]

C6 Constructions must be modeled in a way that prevents the infinite application of rules

10.3.3. Use Case: Validation of the Construction Scope

An exhaustive overview of all analyses which have been defined for SE-DSNL can be found in Appendix C.2.1. Overall, 25 attributions have been specified which define 25 attributes assignments and 15 attribute constraints. Since the analysis specifications are very fine-grained, they take up 744 lines when formatted using the default settings. The accompanying rule implementations in Java (including comments and formatted according to Java coding style guidelines) comprise an additional 1082 lines.

In this section, we will examine three of these analyses more closely to exemplify the application of data-flow analysis in this domain. More specifically, this selection will serve to demonstrate how the notion of flow analysis can be applied to the SE-DSNL approach. While all three specifications focus on the validation of properties of the Construction Scope, it is evident that they can be (and have been) applied to evaluate the Semantic Scope as well with only minimal adaptions. The examples which are presented in the following paragraphs implement restrictions of different complexity. They range from straightforward constraints which require the examination of an object's local properties to more elaborate specifications which rely on the combination of multiple analysis results.

Validating Construction Symbol References

The first example consists of a simple constraint that can be evaluated based solely on locally available information. The goal here is to perform a consistency check on the symbols and mappings of a Construction in accordance with **C4**: If a symbol is set in a mapping (mappings are stored in the containsMappings relationship) as either destination1 or destination2, it must also be included in the list of referenced symbols available through the referencesSymbols association. In the case of missing symbols, an error notification should be raised describing the violation.

The attribution which implements this restriction is depicted in Algorithm 39. It defines an attribute constraint symbols_in_construction_cons which is attached to the Construction class and possesses the severity level error.

The result value is computed by the data-flow rule rule_symbols_in_construction: First the sets of referenced symbols and contained mappings of the Construction are

[31]Precision is the amount of relevant information in relation to all returned information while recall denotes the percentage of information considered to be relevant in comparison to all relevant information.

Algorithm 39 The attribution symbols_in_construction

```
1: Attribution SYMBOLS_IN_CONSTRUCTION
2:    attribute constraint symbols_in_construction_cons : error
3:    extend Construction with
4:       occurrenceOf symbols_in_construction_cons
5:          calculateWith rule_symbols_in_construction
```

```
1: Rule RULE_SYMBOLS_IN_CONSTRUCTION(attrDef, context)
2:    referencesSymbols ⇐ context.referencesSymbols
3:    containsMappings ⇐ context.containsMappings
4:    for all (mapping :  context.containsMappings) do
5:       if (not referencesSymbols→contains(mapping.destination1)) then
6:          missingSymbols ⇐ missingSymbols ∪ mapping.destination1
7:       if (not referencesSymbols→contains(mapping.destination2)) then
8:          missingSymbols ⇐ missingSymbols ∪ mapping.destination2
9:    if (missingSymbols→size != 0) then
10:      context[symbols_in_construction_cons] := 'Missing symbols ' + missingSymbols.name
11:   return (missingSymbols→size == 0)
```

acquired through the respective associations referencesSymbols and containsMappings as defined by the metamodel [2−3]. Then, a loop iterates over each mapping, reading the destination's attribute fields which have been set for this mapping and checking whether they are contained in the referenced symbols of the Construction [4 − 8]. If symbols are found to be missing, a notification is generated for the object which failed the test [9 − 10]. To provide further assistance to the user, this message includes the names of all missing symbols. Finally, the status of the constraint is set to passed (true) or violated (false) depending on the result of the validation [11].

Validating the Construction Generalization Hierarchy

The static semantics for the SE-DSNL language demand that the generalization hierarchy of Constructions is consistent. To fulfill this property, it must be ensured that no cyclic inheritance relationships are present (**C3**). Consequently, if the model does specify cyclic generalizations, a notification should be generated which enables the user to identify the Constructions which are part of a cycle along with a unique identifier for each of these SCCs so that it becomes obvious which Constructions belong to the same cycle.

To achieve this goal, the attribution specification in Algorithm 40 defines an attribute constraint circle_constructions_const which computes the desired output. The constraint relies on a SCC detection carried out for the generalization hierarchy of the Constructions. This information is computed by the attribute circle_construction_values_assign.

The algorithm also lists pseudo code implementations for the data-flow rules

Algorithm 40 The attribution circle_constructions

```
1: Attribution CIRCLE_CONSTRUCTIONS
2:    attribute assignment circle_constructions_values_assign : HashMap
3:       initWith EmptyHashMap
4:    attribute constraint circle_constructions_const : error
5:    extend Construction with
6:       occurrenceOf circle_constructions_values_assign
7:          calculateWith rule_circle_constructions_values_assign
8:       occurrenceOf circle_constructions_const
9:          calculateWith rule_circle_constructions_const
```

```
1: Rule RULE_CIRCLE_CONSTRUCTIONS_VALUES_ASSIGN(attrDef, context)
2:    return circle_generic('all_con_successors_assign', 'Generalization',
   'contains', 'references')
```

```
1: Rule RULE_CIRCLE_CONSTRUCTIONS_CONST(attrDef, context)
2:    sccID ⇐ context[circle_constructions_values_assign]
3:    if (sccID != INIT) then
4:       context[circle_constructions_const] := 'Construction ' +
   context.name ' belongs to SCC ' + sccID
5:       return false
6:    return true
```

rule_circle_constructions_values_assign and rule_circle_constructions_const: The computation of the SCC identifier for the local Construction node is relayed to the method circle_generic. This function (cf. Section 9.2.3) is configured with several analysis-specific parameters: all_con_successors_assign is the attribute responsible for computing the set of transitive successors [2]. In this case, the successors are accessed through the Generalization relationships which, in turn, use the contains association to link back to Constructions through references. Depending on the result of the cycle detection, the attribute constraint rule rule_circle_constructions_const sets an error message which identifies the local context and also includes the unique SCC identifier shared by all Constructions belonging to the current cycle [4].

It should be noted that the separation between an assignment and a constraint attribute necessitates (at least) one additional rule execution for each Construction. However, as will become evident in the next example, this setup enables the reuse of the SCC identifier computed by circle_construction_values_assign.

Validating the Construction Hierarchy

The final example combines the validation of three requirements in a single specification, namely that each Construction has to be part of a single, non-cyclic generalization hierarchy with a unique root element as well as checking for erroneous multiple inheritance relations. This analysis depends on the previously described SCC detection for Constructions and enhances this definition with additional checks.

The goal here is to implement a unified computation of the well-formedness rules **C1, C2** and **C3** as opposed to the previous example which only validates **C3**. For reasons of clarity, only a subset of this attribution is presented and discussed here. The missing parts of the analysis specification can be found in Appendix C.2.2.

Algorithm 41 Data-flow rule for the attribution construction_in_hierarchy

```
1: Rule CONSTRUCTION_IN_HIERARCHY(attrDef, context)
2:     failedObjects ⇐ context[constructions_in_hierarchy_assign]
3:     if (failedObjects→getValue(context) != null) then
4:         roots ⇐ failedObjects→getValue(context)
5:         if (roots→size == 0) then
6:             sccID ⇐ context[circle_constructions_values_assign]
7:             for all (succObj : context[all_gen_successors_objects_assign]) do
8:                 if (sccID == succObj[all_gen_successors_objects_assign]) then
9:                     cycleObjects ⇐ cycleObjects ∪ succObj
10:            context[construction_in_hierarchy_const] ⇐ 'Construction ' +
   context.name + ' is in cycle ' + additions.name
11:        else
12:            context[construction_in_hierarchy_const] ⇐ 'Construction ' +
   context.name + ' is one of multiple roots ' + roots.name
13:        return false
14:    if (context[element_with_single_parent_assign]→size > 1) then
15:        context[construction_in_hierarchy_const] ⇐ 'Too many parents'
16:        return false
17:    return true
```

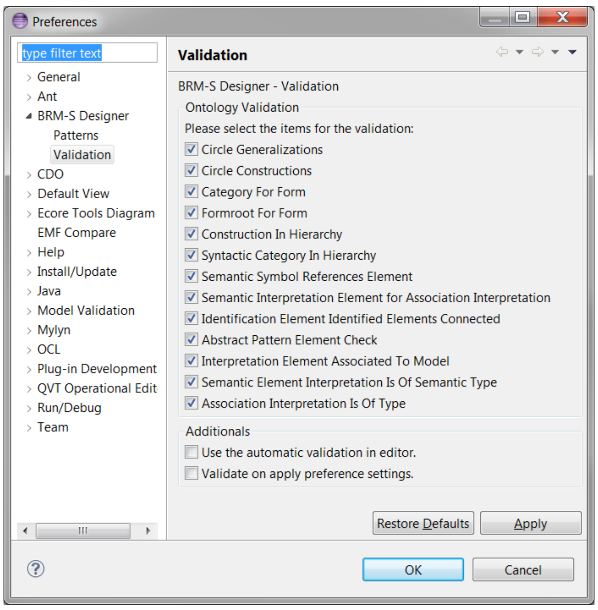
The rule construction_in_hierarchy, shown in Algorithm 41, computes the iteration value of the attribute constraint construction_in_hierarchy_const. This constraint is attached to and therefore evaluated for every Construction in the model. To perform the necessary sanity checks, the map failedObjects is acquired through the attribute constructions_in_hierarchy_assign (note the additional 's') [2]. This data structure is computed once per model at the model's root container, the Domain (cf. Figure 10.28). It associates Constructions which have failed the validation with either an empty list if the Construction is part of a cyclic generalization hierarchy or the list of all Construction root elements if the uniqueness property is violated.

According to the type of the detected problem, a corresponding error message is assembled. This message either indicates which elements constitute the SCC to which this Construction belongs (using the circle_constructions_values_assign as defined in the previous example and the successor analysis as described in Section 9.2.2) [10] or a list of all detected root nodes (which thus violate the unique root element property) [12]. The final check validates whether an element's parents are correctly set by querying and evaluating the attribute element_with_single_parent_assign [14 – 15]. This attribute keeps track of the number of parents a model element has been assigned to based on its contained (contains) Generalization relationships. Note that, because of the generic nature of the metamodel's Reference Scope, this restriction cannot be incorporated into the abstract syntax.

10.3.4. Tooling Integration

The SE-DSNL approach has been implemented in a research prototype. This application makes use of the modeling facilities provided by Eclipse, namely the EMF metamodeling framework and the Graphical Modeling Framework (GMF) which is used to complement the abstract syntax with a graphical DSL editor (cf. Appendix C.2.3). This technological ecosystem facilitates a straightforward integration of analysis capabilities following the approach described in Section 8.3.5. In this section, we will illustrate the technical and functional aspects of this process.

By applying the principles laid out in Figure 8.10, it is possible to extend the SE-DSNL tooling without having to modify any parts of the framework. As a first step, the primary components of the MAF library - namely the plugins MAF-Core and MAF-Models - were added to the existing build configuration. The implementations of the analysis specifications and the technology bridge are combined in a single plugin (brms-validation). This component declares a dependency to the respective SE-DSNL plugins and thus is able to access the graphical editor and the model management facilities.



Validation

BRM-S Designer - Validation

Ontology Validation

Please select the items for the validation:

- ✓ Circle Generalizations
- ✓ Circle Constructions
- ✓ Category For Form
- ✓ Formroot For Form
- ✓ Construction In Hierarchy
- ✓ Syntactic Category In Hierarchy
- ✓ Semantic Symbol References Element
- ✓ Semantic Interpretation Element for Association Interpretation
- ✓ Identification Element Identified Elements Connected
- ✓ Abstract Pattern Element Check
- ✓ Interpretation Element Associated To Model
- ✓ Semantic Element Interpretation Is Of Semantic Type
- ✓ Association Interpretation Is Of Type

Additionals

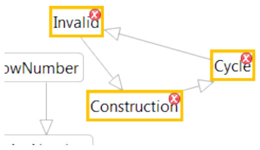
- ☐ Use the automatic validation in editor.
- ☐ Validate on apply preference settings.

Figure 10.30.: The analysis configuration dialog.

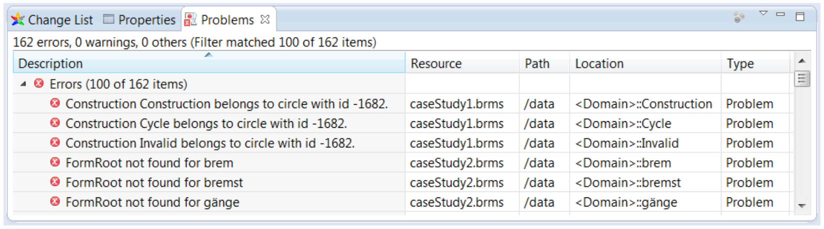
To expose analysis functionality to users, brms-validation contributes preference settings[32] to Eclipse and adds toolbar entries which support the configuration and execution of the validation process. Each time an analysis is triggered, a custom implementation of a MAF model adapter (which is part of the technology bridge)

[32]Eclipse's preference store concept can be used to (de)serialize and store application-specific parameters.

acquires the active model resource from the diagram editor and strips away GMF-specific visualization data to increase performance and reduce memory consumption. In addition to *on-demand* invocations which have to be explicitly triggered by the user, it is also possible to configure the validation services to operate in *live mode*. If this option is selected, the analysis is executed automatically each time the EMF resource reports a change. To reduce the performance impact in live validation mode, the configuration dialog (shown in Figure 10.30) offers the possibility to constrain the evaluation to a relevant subset of the available analyses.



(a) Problem markers in the graphical editor



162 errors, 0 warnings, 0 others (Filter matched 100 of 162 items)

Description	Resource	Path	Location	Type
Errors (100 of 162 items)				
Construction Construction belongs to circle with id -1682.	caseStudy1.brms	/data	<Domain>::Construction	Problem
Construction Cycle belongs to circle with id -1682.	caseStudy1.brms	/data	<Domain>::Cycle	Problem
Construction Invalid belongs to circle with id -1682.	caseStudy1.brms	/data	<Domain>::Invalid	Problem
FormRoot not found for brem	caseStudy2.brms	/data	<Domain>::brem	Problem
FormRoot not found for bremst	caseStudy2.brms	/data	<Domain>::bremst	Problem
FormRoot not found for gänge	caseStudy2.brms	/data	<Domain>::gänge	Problem

(b) Problem markers in the global problem view

Figure 10.31.: Seamless integration of problem reports in the SE-DSNL IDE.

The technology bridge also includes domain-specific implementations of MAF Result Processors which are capable of refining the analysis results. These functions make use of Eclipse's problem indication facilities to generate problem markers for erroneous model elements. These markers are then used to create appropriate visualizations for the detected constraint violations in both the graphical editor and in Eclipse's global problem view: The GMF editor displays problem markers as symbols in the context of the respective diagram elements (cf. Figure 10.31(a)) while the problem view shows a list of the markers attached to the currently active model resource (cf. Figure 10.31(b)). It is possible to select an entry in the problem view and directly navigate to its counterpart in the graphical editor.

10.3.5. Evaluation of the Use Cases

The SE-DSNL prototype ships with two case studies. They consist of large models which simulate the requirements of real-world application scenarios. It can therefore be assumed that these case studies also represent a solid basis for the evaluation of the developed analyses. Information about the instantiation and solving processes has been recorded using MAF's inbuilt statistical evaluation facilities. Table 10.9 shows the acquired results for the execution of all available analyses for both example models. Conversely, Table 10.10 lists the statistical recordings for the evaluation of a single analysis, constructions_in_hierarchy. It should be noted that the latter case also includes the computation of implicitly required information such as the successor analysis for the Generalization hierarchy.

It can be seen that the models contain a large amount of objects: 3324 and 2054 model elements for caseStudy1.brms and caseStudy2.brms respectively. Table 10.9

File	Model elements	Attribute instances	Rule exec.	Cons. passed	Cons. violated	Time
caseStudy1.brms	3324	4359	4377	1456	37	534.1ms
caseStudy2.brms	2054	2685	2685	864	159	378.1ms

Table 10.9.: Results for all attributions.

File	Elements	Instances	Exec.	Cons. passed	Cons. violated	Time
caseStudy1.brms	3324	1186	1201	273	29	121.9ms
caseStudy2.brms	2054	413	413	103	0	55.0ms

Table 10.10.: Results for attribution constructions_in_hierarchy.

indicates that, if all defined analyses are executed, the instantiation of the attributions yields 4359 attribute instances for the first and 2685 instances for the second model. By definition, the amount of instances also represents the lower bound for the execution of iteration rules since each attribute must be computed at least once. For caseStudy1.brms, the number of rule invocations (4377) is only slightly higher than the amount of attribute instances. Since the execution of initialization rules is excluded from this number, this indicates that an iterative fixed-point computation must have taken place. For the second model caseStudy2.brms, each data-flow rule was executed only once, suggesting that no cyclic dependencies exist between the contained elements.

The examination of the statistical data leads to the conclusion that the constraint attributes (which represent the final analysis results) amount to about roughly a third of all instances with the remaining attributes realizing preparatory functions such as SCC detection. Considering that the case studies are officially shipped with the SE-DSNL tooling, it is perhaps surprising that a substantial number of violations were discovered. The detected problems amount to 2,48% of the total number of checked constraints for the first and 15.54% for the second example.

Table 10.11 lists the incidences of the failed attribute constraints: In both models, violations have been detected for formroot_for_form_const (a sanity check performed on the Syntactic Scope of the SE-DNSL model). This result indicates a number of dangling Form elements which are not contained in a FormRoot. Another check which fails for both examples is semantic_symbol_references_element_const. This problem type points to a faulty structure in the Semantic Scope. This error is however much more prevalent in the second model. Finally, circle_constructions_const and construction_in_hierarchy _const are checks that failed only for the first example. The former constraint indicates the presence of cyclic Generalization hierarchies while - according to the generated information messages - the latter was able to identify irregularities in the form of multiple root elements.

For obvious reasons, the time required for the completion of an analysis is an important factor which affects the usability of the developed applications. This is especially true for domains like SE-DSNL where large-scale models are common and

Analysis	**Amount**	**Description**
caseStudy1.brms		
formroot_for_form_const	1	Validates whether each Form belongs to a FormRoot
circle_constructions_const	2	Checks whether Constructions contain a cycle
semantic_symbol_references _element_const	5	Validates whether all SemanticSymbols possess a valid referencesElement value
construction_in_hierarchy _const	29	Validates whether each Construction is part of a hierarchy, does not belong to a cycle and if there exists only one root Construction
caseStudy2.brms		
formroot_for_form_const	5	Validates whether each Form belongs to a FormRoot
semantic_symbol_references _element_const	154	Validates whether all SemanticSymbols possess a valid referencesElement value

Table 10.11.: Violated constraints for caseStudy1.brms and caseStudy2.brms.

analyses have to be executed repeatedly to provide immediate feedback to the user during the modeling process. Not surprisingly, the statistical evaluation shows that the worst case consists of the evaluation of all 25 attributions for the 3324 model elements of caseStudy1.brms. Overall, this process takes about half a second. On the other hand, the constrained evaluation of the same model (which nevertheless computes 1186 separate result values) takes about 120 milliseconds.

10.3.6. Summary and Discussion

In this case study, we introduced the SE-DSNL approach and discussed several flow-based analyses which implement sanity checks for the model-based domain-specific language. The specification of these analyses and the development of the corresponding tooling extensions was supervised by the author of the SE-DSNL technique. For this reason, this case study represents an opportunity for an evaluation of the application of DFA in what constitutes a typical usage scenario: Based on an informal definition of well-formedness rules, analyses had to be specified and the tooling environment had to be augmented with the required capabilities. This matter concerns the extension of existing program modules in a non-intrusive manner as well as a seamless integration of analysis configuration and execution functionality and a suitable visualization of the results. Furthermore, the comparatively large models in this domain facilitate an evaluation of the performance aspects of the DFA solving algorithms implemented in the Model Analysis Framework.

Section 10.3.3 detailed three specific use cases chosen from the set of all analyses which have been defined for the SE-DSNL domain (listed in Appendix C.2.1). These examples range from simple constraints which can be computed from an element's local properties to more complex, incremental specifications which rely on results from other analyses and reuse templates from the DFA standard library. Because the objective of this case study represents a typical validation scenario, the final result of each analysis always consists of a set of attribute constraints. While some restrictions, such as **C4** could also be implemented using canonical constraint languages such as OCL, others such as **C3** rely on the principles of information propagation and fixed-point computation. It was demonstrated that the flow analysis technique supports the implementation of both simple and complex rules and thus provides a unified interface for analysis developers as well as for users. From the list of constraints provided in Section 10.3.2, it can be deduced that analysis specifications for the different scopes of the language are often very similar in nature. For example, both the Syntactic Scope and the Semantic Scope require non-cyclic inheritance hierarchies. However, at the same time, they either explicitly allow or prohibit multiple inheritance for Generalizations. The presented analyses limit the computational overhead by relying on already computed results. Reuse on the implementation level could still be improved, for example through a parameterization of the data-flow rules as demonstrated in Section 10.1.

Unsurprisingly, it is not possible to implement all restrictions of the target domain using static analysis methods as some of the language's semantics define conceptual rather than structural constraints. This includes for example **S6** and **C5** which can be interpreted as guidelines for language engineers. Others - such as **C6** - involve dynamic properties of the language that cannot be verified using static methods.

The evaluation of the official case studies packaged with the SE-DSNL framework yielded some interesting insights: In general, there were almost no violations of the *no cyclic definitions* constraints, meaning that fixed-point computation was only necessary in the rare occasion of an erroneous specification. In fact, most of the detected errors belong to the category of basic syntactic problems. In this context, it is important to emphasize that the confirmation of the correctness of a model is just as relevant as the detection of errors. It can also be assumed that, since the supplied models represent fully implemented use cases which have already been put to the test in various NLP scenarios, the absence of major problems is not surprising. Obviously, the situation will be different for inexperienced users or when new models are created from scratch. In these cases, the incremental validation of the constraints during the modeling process can be very helpful since the detection of problems during early stages of the design phase is likely to prevent more complex problems which might arise from the erroneous definitions in later stages.

We will now summarize our findings:

Implementation effort

The SE-DSNL framework is built upon the Eclipse platform and makes use of the EMF and GMF facilities to implement a domain-specific language. This setup provided an excellent starting point for the integration of the Model Analysis Framework: No existing code had to be modified in order to extend the tool chain with analysis capabilities. Configuration dialogs and visualization of the analysis results are fully integrated with the IDE. The notification subsystem which employs Eclipse's problem marker infrastructure to provide feedback to the user was contributed to the Model Analysis Framework by the author of the SE-DSNL approach.

Classification
Since this case study automates the checking of restrictions which were originally stated in an informal fashion, it represents a typical validation scenario for the DFA approach. The final results consist of attribute constraints which rely on intermediate computations to generate exhaustive information about detected problems: In addition to the status (passed/violated), each constraint indicates the violation type (error/warning/information) and outputs a message which details the respective findings.

Reuse
The analyses make use of the functions described in the standard library for control-flow graphs (cf. Section 9.2) which have been adapted to the SE-DSNL domain. The computation of transitive predecessor/successor sets enables the detection of multiple root elements in Generalization hierarchies and represents the basis for the identification of cyclic paths. The aspect of reuse can also be found in the domain itself as some constraints can be applied to different scopes of the SE-DSNL DSL with minimal modifications.

Usability
The seamless integration with the SE-DNSL tooling enables language engineers to focus on their respective tasks. Since the result visualization makes use of the corresponding Eclipse facilities, users can rely on their previous experience with this IDE while benefiting from clear and easily accessible feedback. A very useful feature in this context is the configurable live analysis mode which automatically updates the validation status each time the state of the model changes.

Performance
The analyses were applied to two example models and the characteristics of their execution were recorded using MAF's statistical evaluation facilities (cf. Section 10.3.5). These measurements as well as the experiences reported by the users indicate that this approach represents a valuable addition for language engineers. However, the typically very large SE-DNSL models have been identified as an inherent challenge of this domain and as a result, the usage of the live validation mode is typically restricted to a subset of the available analyses.

During the review of the implemented analyses, we identified several starting points for future work. For example, it would be possible to extend the current functionality by implementing a more exhaustive validation of the restrictions listed in Section 10.3.2. Additionally, we intend to provide further assistance which enables users to not only validate but also improve the quality of the modeled artifacts.

Implementation of constraint (S5)
This constraint states that the type of an Association must be part of a different Generalization hierarchy than the Association's source and target elements. To implement this constraint, it is necessary to compute a unique identifier for each inheritance graph and assign this value to all participating objects. For this purpose, it would be sufficient to generate the hash value of the set of all elements which belong to a specific hierarchy. The participating elements can be computed by creating the union of the transitive Generalization successors and predecessors. Finally, a constraint defined in the context of Associations can compare the identifier of the Association's hierarchy to the identifiers of its referenced elements.

Improve precision/recall (C5)
Achieving a satisfactory balance between precision and recall is an often encountered challenge in the area of information retrieval. In the case of NLP-enabled information extraction, the quality of the modeled knowledge heavily depends on the respective domain and the intended use cases. For this reason, a majority of the responsibility rests with the knowledge engineer who encodes domain-specific information using the SE-DSNL approach. Although it is difficult to assess the quality automatically, it would be possible to implement a set of heuristics which are able to assist the user in this task. One example would be the measurement of the granularity on a structural level. A concrete implementation could yield the average depth of Construction hierarchies and compute whether they form well-balanced trees. Depending on the desired level of detail (fine-grained or coarse) this would enable the modeler to assess the quality of the model with respect to the current requirements.

Heuristics for infinite loop detection (C6)
Infinite loops are a result of the interpretation of the SE-DSNL models according to their dynamic semantics. For this reason, a reliable detection of these cases cannot be accomplished using static analysis. However, it would nevertheless be possible to compute a maximal approximation by taking into account every valid rule application. This way, potential problems can be identified and indicated to the user who can then examine the potentially problematic instances. Extending the SE-DSNL framework with the ability to mark Construction elements as relevant or ignore them based on a manual review can help to improve the precision of this analysis.

Further changes can be made to improve the structure of the attributions and the performance of the analysis:

- One could argue that the level of detail of the analysis specifications - mostly one attribute per attribution - may in fact be too fine-grained. A viable alternative would be to cluster the attributions by scope and only execute the analysis collection for the scope in which the user is currently active. This solution would not only reduce the maintenance effort for developers but also provide users with the required information for their current working context while at the same time improving the performance of the analysis.

- The performance issues which arise from the large-scale models in this domain can also be addressed by other means. For example, instead of analyzing the whole model, only the elements that have been modified by a user interaction can be submitted for reevaluation. The dynamic dependency discovery feature of the DFA algorithms ensures that all secondary attributes which are relevant to the evaluation of the requested results are automatically computed as well. Furthermore, it would be possible to execute the analyses in a background process to ensure that the validation does not interfere with the modeling process. In this case, the results would be displayed in a time-delayed fashion once the analysis has completed.

10.4. Case Study: AUTOSAR

In this case study, we will present a use case in the context of the development of modern automotive systems. Due to the increasing computerization of cars, the physical limitations of current embedded computing platforms become more and more evident. A promising solution to this problem exists in the form of multicore systems which provide more computational power while consuming less energy. However, due to the inherent complexity of parallelized software, new challenges arise that have to be addressed by car manufacturers and component suppliers alike. For this reason, developers require new methods and tools which support them in the task of designing and implementing software for embedded multicore systems.

This case study is based on the research work detailed in the peer-reviewed paper [Kie+14] and the master thesis [Min12]. Both publications focus on the validation of AUTOSAR models with the stated goal of optimizing the system design for parallelized hardware. An important step in this process consists of the examination of data dependencies between the modeled components, the so-called RunnableEntities. The goal is to provide feedback to developers about the validity of their architectural design and to offer semi-automatic problem resolution methods for identified issues. The proposed approach for flow-based model analysis represents an integral part of the methodology described in this section.

10.4.1. Introduction and Motivation

Today, manufacturers in the automotive domain have to deal with increasing expectations from customers concerning the availability of safety and entertainment features. An additional complication exists in the fact that the manufacturing process itself is more and more characterized by a fine-grained division of labor: Car manufacturers assemble their products from components that they buy from different vendors and which must be configured to work together properly. Because of the large number of components and the increasingly complex interactions between them, this has become a very challenging task. One of the industry's responses consists of the AUTomotive Open System ARchitecture (AUTOSAR) initiative. The stated intent of this cooperative effort - which involves many leading car manufacturers, component suppliers and providers of software tools - is to standardize interfaces and development methodologies in the automotive domain.

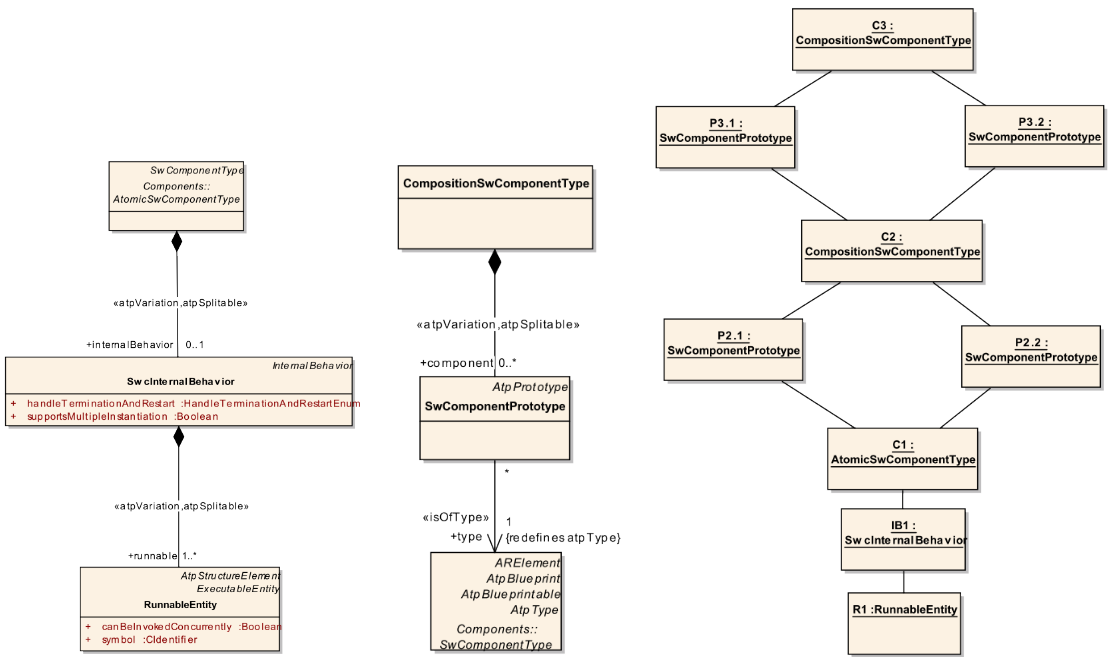
The adoption of the AUTOSAR methodology - and its model-based domain-specific language - provides an opportunity to address the challenges which result from the increasing demands of non-mechanical (i.e. software) functionality. As automotive software systems become more complex, more processing power is needed to satisfy the higher requirements. Because *"single core architectures are approaching their limits concerning for example clock speed, memory speed and temperature [the] system manufacturers are trying to reduce the number of Electronic Control Units (ECU) while increasing the system functionality"* [Min12]. For cost reasons and because of physical limitations, this goal must be achieved at lower rates of energy consumption than is currently possible. An obvious solution to this problem lies in the switch from single to multicore systems. Unfortunately, this step also raises new issues as the derivation of a suitable parallelized scheduling for the functional software entities in AUTOSAR models is a non-trivial task. More specifically, it must be ensured that the development models specify precise and consistent execution orders. In a subsequent step, these can then be used to compute a scheduling which balances the workload for multiple processing cores.

AUTOSAR

The AUTOSAR initiative[33] was *"founded in 2003 by major OEMs and Tier1 suppliers and now includes a large number of automotive, electronics, semiconductor, hard- and software companies. AUTOSAR aims at facilitating the re-use of soft- and hardware components between different vehicle platforms, OEMs and suppliers"*. This is achieved through a *"methodology that supports a distributed, function-driven development process and standardizes the software-architecture for each ECU in such a system"* [Fen+06]. A central element of AUTOSAR is the Virtual Function Bus (VFB) which decouples the software components from the hardware infrastructure through standardized interfaces. In the following, we will discuss aspects of AUTOSAR which are of relevance to the implementation of this use case.

The smallest functional unit defined by the AUTOSAR metamodel is the RunnableEntity. This type represents an executable piece of code and is a subclass of ExecutableEntity. A runnable is part of a software component (SwComponentType, cf. Figure 10.32(a)). It is important to note that the same RunnableEntity can have multiple occurrences in a single model: As can be seen in Figure 10.32(b), a composed component (CompositionSwComponentType) may reference arbitrary SwComponentTypes. The implications of this principle are illustrated in Figure 10.32(c). The

[33] `http://www.autosar.org`



(a) A RunnableEntity is part of an InternalBehavior, which in turn is part of a component

(b) A CompositionSwComponentType can include several instances of the same component

(c) Example with four instances of the same RunnableEntity

Figure 10.32.: Software components and runnable entities in AUTOSAR [Min12].

CompositionSwComponentType C2 itself is used in two contexts (P3.1 and P3.2). Since C2 in turn has two references to the AtomicSwComponentType C1 which contains the runnable R1, there are four “instances” of R1 overall.

The properties of the communication between runnables can be specified in more detail using different types of constraints. For example, AgeConstraints (AC) can be used to impose a limit on the age of a data element. This is necessary if write and read accesses span multiple computing cycles. In this case, an AC can be defined which explicitly states that reading a value computed in a previous cycle is legal. An ExecutionOrderConstraint (EOC), on the other hand, imposes a specific execution order by annotating an ExecutableEntity with a list of required successor entities. Both constraint types are relevant to this case study as both affect whether a concrete scheduling for runnables is considered to be valid.

Multicore Development for AUTOSAR models

The computation of correct schedulings for RunnableEntities requires knowledge about valid execution sequences. In a multicore environment, it is often preferable to allow for a certain amount of flexibility as this provides the scheduling algorithm with more options when assigning functional units to processing cores: [Kie+14] states that developers have to *“preserve as much freedom as possible and simulta-*

neously prevent the system from entering problematic states that cause, e.g., race conditions, data inconsistencies or dead locks". As mentioned, in AUTOSAR, this can be achieved through the use of ExecutionOrderConstraints: Rather than imposing a rigid sequence for the execution of entities, these elements represent a set of conditions which must be met by each valid scheduling. At the same time, these constraints can be used to validate the current system design, for example by checking for potentially problematic cyclic data dependencies. Furthermore, it is possible to improve an existing design by making constraints explicit whose existence can be derived from the model's structure. This enables the scheduler to explore a more focused set of candidates to find valid and well-balanced execution sequences.

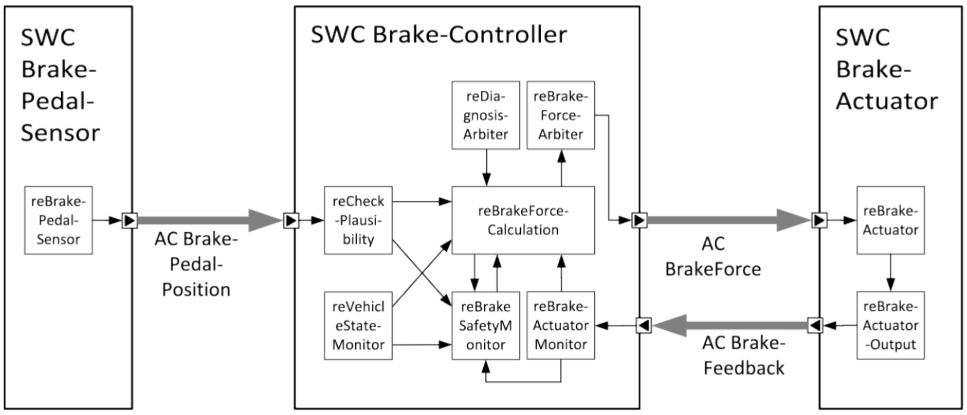
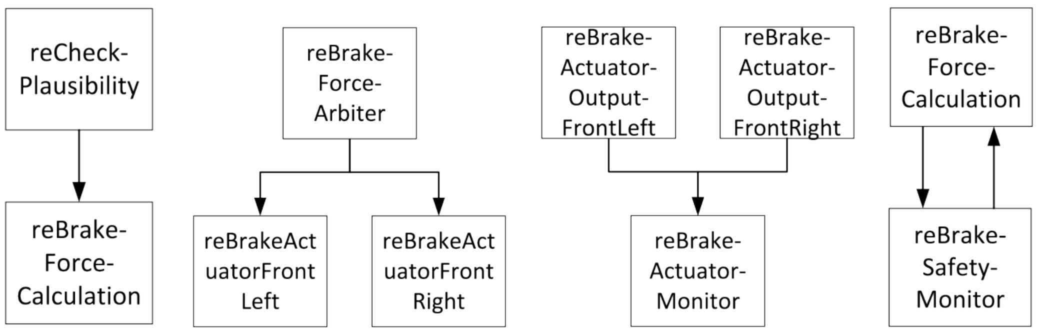


Figure 10.33.: Runnable entities of the TIMMO breaking system [Min12].

The authors propose multiple constraint detection and resolution techniques which are based on an in-depth analysis of ExecutionOrderConstraints and AgeConstraints in AUTOSAR models. The method which we will examine more closely in this case study deals with the validation and derivation of constraints based on an evaluation of the declared data dependencies. [Kie+14] states that the *"goals are to discover design weaknesses, to automatically solve trivial pitfalls, to support the elimination of the remaining conflicts (mainly cycles) and to write back the modifications to the model"*. The application scenario described in this section will be exemplified in the context of an example model which has been developed by the TIMMO project[34]. This model, shown in Figure 10.33, describes the architecture of a breaking system. It defines three main software components - a brake pedal sensor, a brake controller and a brake actuator - each of which contains one or more runnables. The components exchange data using AssemblyConnectors, for example BrakePedal-Position which forwards sensor data to the controller. However, information is also exchanged on the RunnableEntity level: The connector BrakeForce supplies an input value to the runnable reBrakeActuator which processes this information and relays it to reBrakeActuatorOutput which in turn provides feedback to reBrakeActuatorMonitoring.

[Min12] emphasizes the importance of data dependencies for the determination of schedulings for parallelized executions: *"[M]ost challenges of multi core systems are*

[34] http://www.timmo-2-use.org


reCheck-Plausibility
reBrake-Force-Calculation
reBrake-Force-Arbiter
reBrakeActuatorFrontLeft
reBrakeActuatorFrontRight
reBrake-Actuator-Output-FrontLeft
reBrake-Actuator-Output-FrontRight
reBrake-Actuator-Monitor
reBrake-Force-Calculation
reBrake-Safety-Monitor

(a) reCheckPlausibility writes a variable that is read by reBrakeForceCalculation

(b) reBrakeForceArbiter writes a variable that is read by reBrakeActuatorFrontLeft and reBrakeActuatorFrontRight

(c) reBrakeActuatorFrontLeft writes a variable that is also written by reBrakeActuatorFrontRight and read by reBrakeActuatorMonitor

(d) reBrakeForceCalculation and reBrakeActuatorMonitor exchange data in both directions

Figure 10.34.: Data dependencies in the running example [Min12].

related to the interaction between the (runnable) entities running on different cores" as *"data dependencies can add constraints on the accepted execution orders of the runnable entities, because any data has to be produced before it can be consumed"*. Problems that are especially relevant the multicore context include *"data races and deadlocks. Additionally multi core development has to cope with non determinism and load balancing"*. Figure 10.34 depicts different kinds of data dependencies between the RunnableEntities. In the case shown in Figure 10.34(a), reCheckPlausibility must be finished before reBrakeForceCalculation is started, if the latter requires the value computed in the current cycle. Similar semantics apply if multiple entities are involved as is the case in Figures 10.34(b) and 10.34(c). Circular dependencies, as depicted in Figure 10.34(d), indicate conflicting constraints. These can, for example, be resolved by introducing AgeConstraints which explicitly allow entities to access data produced in previous computing cycles. Alternatively, EOCs can be defined which enforce a specific execution order.

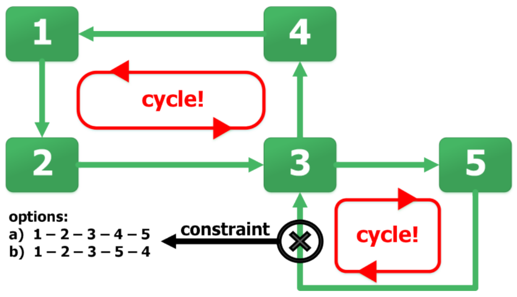


Figure 10.35.: Conflict resolution for cyclic data dependencies [Kie+14].

This principle is illustrated in Figure 10.35 for five runnable entities (REI). The model *"shows a conflict example where the calculated output from "REI 3" is a neces-*

sary input for the calculation of "REI 5" and vice versa. This cycle has to be broken up via a constraint that allows the transfered data to be as old as one computing cycle or that imposes an execution order on the involved REIs" [Kie+14].

The information which can be obtained through an analysis of ExecutionOrderConstraints can be supplemented by an evaluation of existing AgeConstraints: If a read and a write operation are both required to occur in the same cycle, it can be deduced that the write operation must be carried out before the read access. If the AgeConstraint states that the read operation may use data which has been computed in a previous cycle, the operations can be carried out in any order. The same is true if no constraint has been specified for the data age, although it is generally safer to assume that accesses must happen in the same cycle. Alternatively, an advisory can be generated which prompts the user to resolve this situation.

Further methods for the validation of ExecutionOrderConstraints include the analysis of control-flow dependencies and - if available - entity-to-task mappings. Although these mappings impose a strict ordering on the tasks, they should still be validated to check whether the control-flow conforms to the declared data dependencies.

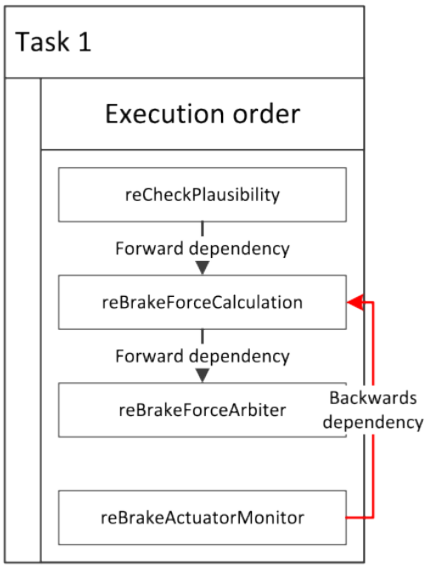


Figure 10.36.: Dependencies inside a task [Min12].

The task mapping in Figure 10.36 defines a sequential order for the execution of runnables. Data dependencies between *"reCheckPlausibility and reBrakeForceCalculation and also between reBrakeForceCalculation and reBrakeForceArbiter are forward dependencies, because in one computing cycle, the variables are first written then read. The data dependency between reBrakeActuatorMonitor and reBrakeForceCalculation is a backward dependency, because in one computing cycle the variable is first read then written."* [Min12]. It is important to note that, in the latter case, the read and write operations for a specific instance of the respective data object take place in different computing cycles: The read access in iteration n returns the value set by the write operation in iteration $n-1$. As mentioned above, this case requires the definition of an AgeConstraint. In other words, this is considered to be an *intentional backward dependency.*

10.4.2. Use Case: Backward Dependency Analysis

Precomputation and Caching of Model Relationships

Due to the complex nature of the AUTOSAR modeling language, extracting information from the model is a difficult and time consuming task. For example, since many of the relationships between elements are unidirectional, navigation in the opposite direction requires a traversal of a large amount of objects to identify the respective source. Furthermore, the structure of the metamodel is very detailed (cf. Figure 10.32). It is therefore often impossible to access the necessary information with a single navigational statement. In conclusion, because models in the AUTOSAR domain tend to be very large and the iterative evaluation of data-flow attributes may result in many requests to read data from the model, these complex and costly operations can present a problem for the intended use case.

To address this issue, the authors of the analysis approach propose the execution of a pre-analysis step which extracts the required information and stores it in an object cache for quick access. The MAF-based implementation realizes this functionality through a set of storage maps which are automatically filled before an analysis is executed. This approach also addresses another problem: A single RunnableEntity may have multiple logical occurrences in the AUTOSAR model (cf. Figure 10.32(c)) and therefore may possess multiple contexts with respect to the dependency analysis. It is therefore necessary to compute a separate DFA result for each of these occurrences. This problem is comparable to the situation in the JWT case study where the same subprocess can be referenced at different locations in the parent process (cf. Section 10.1.1). For this reason, the quick access maps enable data-flow rules executed for runnables to query all contexts in which the respective element appears.

The author of [Min12] lists the following properties which have to be computed ahead of time to improve the performance of the analysis:

- RunnableEntities store data accesses to variables. From this information, a map can be generated which links variables to the runnables which perform read or write operations on the respective entry.

- The ExecutionOrderConstraints for RunnableEntities may be organized in arbitrary packages. For this reason, the whole model must be traversed to build a map which stores the successors for each RunnableEntity.

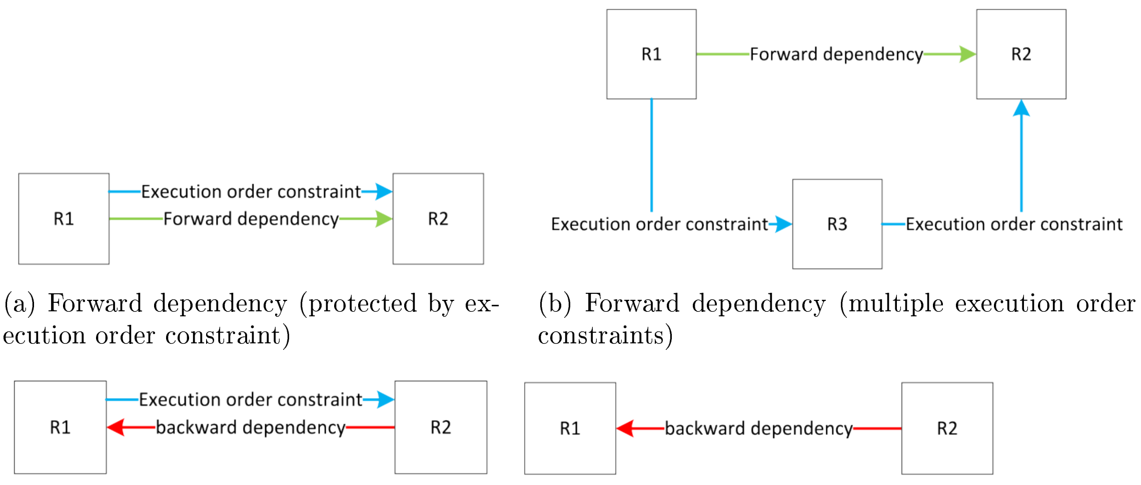
- RTEEvents define triggering frequencies for runnables. This information is required to assess the status of AgeConstraints. For this purpose, the links between runnables and their timing events must be evaluated and made available to the analysis.

- Finally, a reversed access map must be generated which connects DataPrototypes to AgeConstraints.

Analysis of Potential Backward Dependencies

As stated above, the evaluation of ExecutionOrderConstraints involves the analysis of data dependencies and AgeConstraints. [Min12] states that *"the conversion of data dependencies to execution sequence constraints corresponds to an analysis finding data dependencies for which data age constraints might be violated for the given set of execution sequence constraints"*. The importance of the evaluation of RunnableEntities is emphasized in particular as *"cyclic dependencies at component level do not necessarily lead to cyclic dependencies at runnable entity level"* and therefore *"there has to be an analysis on runnable entity level, rather than only an analysis at component level"*.

In the first step of the analysis, the outgoing data dependencies of a runnable must be read from the object cache. Then, the data prototypes of the respective output variables (VariableDataPrototype) can be acquired. Using this information, a set of RunnableEntities which perform read accesses on these variables can be built. By combining all of these results, it is possible to organize the runnables according to their read/write relationships. Finally, these relationships can be classified as either forward or (potential) backward dependencies.


R1
Execution order constraint
Forward dependency
R2
R1
Forward dependency
R2
Execution order constraint
R3
Execution order constraint
R1
Execution order constraint
backward dependency
R2
R1
backward dependency
R2
(a) Forward dependency (protected by execution order constraint)

(b) Forward dependency (multiple execution order constraints)

(c) Backward dependency

(d) Potential backward dependency

Figure 10.37.: Data dependency types [Min12].

Figure 10.37 depicts different configurations of data accesses. The variable access shown in Figure 10.37(a) imposes a *forward dependency* as the accompanying EOC prohibits R1 from being executed after R2. This is also true if the execution order of runnables is specified indirectly as is the case in Figure 10.37(b). To correctly assess this scenario, it is necessary to compute all possible successors of a runnable. The example shown in Figure 10.37(c) specifies a read access by R1 that happens before the variable value is set by R2. Because of the ExecutionOrderConstraint which explicitly states that R1 must be executed before R2, this access has to be classified as a *backward dependency* which must be resolved using an AgeConstraint. The situation in Figure 10.37(d) is somewhat similar although, in this case, no constraint has been defined. Since no information is available about the execution order of R1 and R2, this access denotes a *potential backward dependency*.

Algorithm 42 The attribution obligatory_successors

```
1: Attribution OBLIGATORY_SUCCESSORS
2:    attribute assignment obligatorySuccessors : Map(EEI, Set(EEI))
3:       initWith ∅
4:    extend RunnableEntity with
5:       occurrenceOf obligatorySuccessors
6:          calculateWith runnableentity_obligatorySuccessors
```

```
1: Rule RUNNABLEENTITY_OBLIGATORYSUCCESSORS(attrDef, context)
2:    executableToInstanceMap ⇐ storageMap.get("executableToInstanceMap")
3:    directSuccessorsMap ⇐ storageMap.get("directSuccessorsMap")
4:    for all (eei : executableToInstancesMap.get(localObject)) do
5:       directSuccessors ⇐ directSuccessorsMap.get(eei)
6:       successorsForInstance ⇐ directSuccessors
7:       for all (successor : directSuccessors) do
8:          successorRunnable ⇐ successor.getRunnableEntity()
9:          successorResult ⇐ successorRunnable[obligatorySuccessors]
10:         successorsForInstance ⇐
11:            successorsForInstance ∪ successorResult.get(successor)
12:      resultMap.put(eei, successorsForInstance)
13:   return resultMap
```

The attribution listed in Algorithm 42 computes the set of all successors for a RunnableEntity as a prerequisite for the identification of transitive forward dependencies (cf. Figure 10.37(b)). The analysis is carried out by the data-flow attribute obligatorySuccessors which computes a result for each runnable. As mentioned above, this process is complicated by the fact that a single RE can have multiple occurrences in an AUTOSAR model, each of which has to be considered separately. For this reason, the data-flow rule returns a map which connects each instance of the current runnable to a set of entity instances which are its immediate or indirect successors according to the available ExecutionOrderConstraints. Each context in which a runnable occurs is represented as a ExecutableEntityInstance (EEI), a wrapper class provided by the pre-analysis step. This type identifies specific RunnableEntity "instances" based on their respective component containment hierarchies.

In lines [2 − 3], the rule runnableentity_obligatorySuccessors first acquires the precomputed access maps and subsequently iterates over all EEIs of the current RunnableEntity [4 − 12]. The direct successors of the respective ExecutableEntityInstance are read and added to the overall successor list for this particular instance [5 − 6]. The recursive computation is implemented by the loop in lines [7 − 11] which iterates over the direct successors and queries their data-flow value for the attribute obligatorySuccessors and adds it to the current instance's results. Finally, the new value is stored in the result map [12].

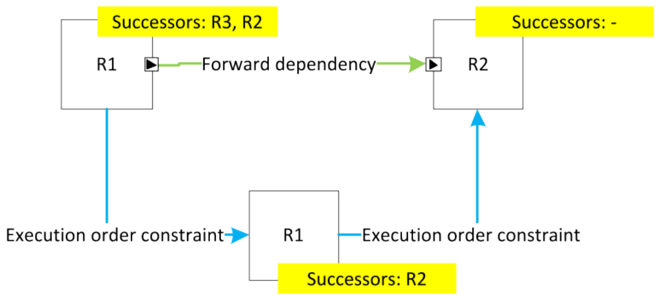


Figure 10.38.: Obligatory successors for RE instances [Min12].

The results of the application of this analysis can be seen in Figure 10.38. The diagram contains three ExecutableEntityInstances, two for the runnable R1 and one for R2. Each of these occurrences lists the computed successors (the entry at the top left instance contains an error: R3 should be R1). This information can now be used to classify the data dependencies. If a runnable writes to a variable that is read by a succeeding runnable, this access represents a forward dependency. If the RunnableEntity which performs the read operation is not a successor, it is a (potential) backward dependency.

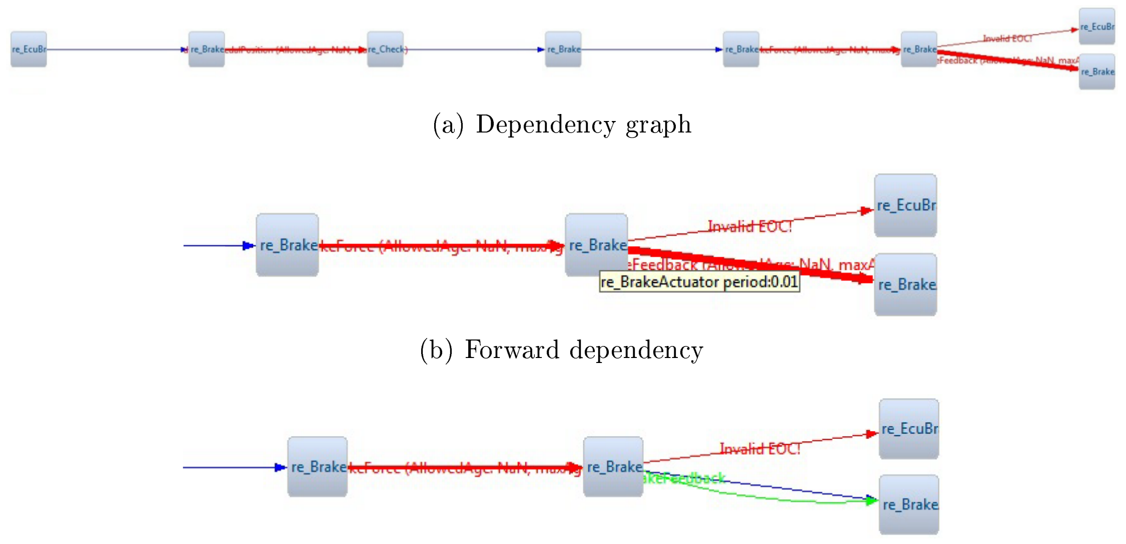
Algorithm 43 The attribution backward_dependencies

```
1: Attribution BACKWARD_DEPENDENCIES
2:     attribute assignment backwardDependencies : Map(EEI, Set(DataExec))
3:         initWith $\emptyset$
4:     extend RunnableEntity with
5:         occurrenceOf backwardDependencies
6:             calculateWith runnableentity_backwardDependencies
```

```
1: Rule RUNNABLEENTITY_BACKWARDDEPENDENCIES(attrDef, context)
2:     executableToInstanceMap ⇐ storageMap.get("executableToInstanceMap")
3:     outgoingDependenciesMap ⇐ storageMap.get("outgoingDependenciesMap")
4:     for all (eei : executableToInstancesMap.get(localObject)) do
5:         runnableSuccessors ⇐ eei.getRunnableEntity()[obligatorySuccessors]
6:         eeiSuccessors ⇐ runnableSuccessors.get(eei)
7:         for all (variableAccess : outgoingDependenciesMap.get(eei)) do
8:             if (not eeiSuccessors→contains(variableAccess.target)) then
9:                 backwardDep ⇐ backwardDep $\cup$ variableAccess
10:        resultMap.put(eei, backwardDep)
11:    return resultMap
```

This classification of data accesses is carried out by the attribution backward_dependencies shown in Algorithm 43. Again, the data-flow rule first acquires the precomputed access maps [2−3]. In this case, this includes outgoingDependenciesMap which stores the variable accesses of ExecutableEntityInstances. Then, the loop in lines [4 − 10] processes the EEIs of the current RunnableEntity. Lines [5 − 6] retrieve the result map computed by obligatorySuccessors and extract the information for the current entity instance. It is now possible to process the outgoing dependencies by iterating over the relevant variable accesses [7 − 9]. If the target of the variable access is not a successor (which would indicate a forward dependency), the access is classified as a (potential) backward dependency [9]. Finally, the identified backward dependencies are stored in the overall result map [10].

Intentional backward dependencies can now be recognized by examining which AgeConstraints explicitly allow access to data computed in previous cycles. If no matching AC can be found, the backward dependency must be considered unintentional and should be reported to the user for manual problem resolution.



(a) Dependency graph

(b) Forward dependency

(c) Intentional backward dependency

Figure 10.39.: Visualized results of the dependency analysis [Min12].

Figure 10.39(a) shows an excerpt of the results computed for the breaking system example. In this figure, potential backward dependencies are plotted as red arrows. The problem resolution system can now present the findings to the user and semi-automatically resolve these problems by generating appropriate constraints. For example, the highlighted backward dependency in Figure 10.39(b) can be transformed into a forward dependency by adding a corresponding EOC (cf. Figure 10.39(c)).

10.4.3. Tooling Integration

It is possible to implement the presented use case in the form of a fully integrated tooling environment. This can, for example, be achieved by extending an existing AUTOSAR development environment such as the Eclipse-based ARTOP[35] platform.

[35]http://www.artop.org/

The thesis assignment for [Min12] included the realization of a proof-of-concept prototype based on MAF and the visualization framework Eclipse Zest[36]. The prototype is able to evaluate dependencies in AUTOSAR models using the described analyses. The results are presented in a diagram editor which enables developers to interactively apply the derived conflict resolutions. For this purpose, the tool provides options for the creation of new EOCs to enforce forward dependency relationships and the generation of suitable AgeConstraints which mark a backward dependency as intentional. Once all problems have been resolved, the modifications can be applied to the original AUTOSAR model. In conclusion, this approach supports the extension of existing automotive development platforms with the functionality required for a goal-oriented validation and refinement of existing system designs. On a technical level, integration with third-party tools is ensured by storing the results of the analysis in the AUTOSAR format, thereby making them available to any application which supports this standard.

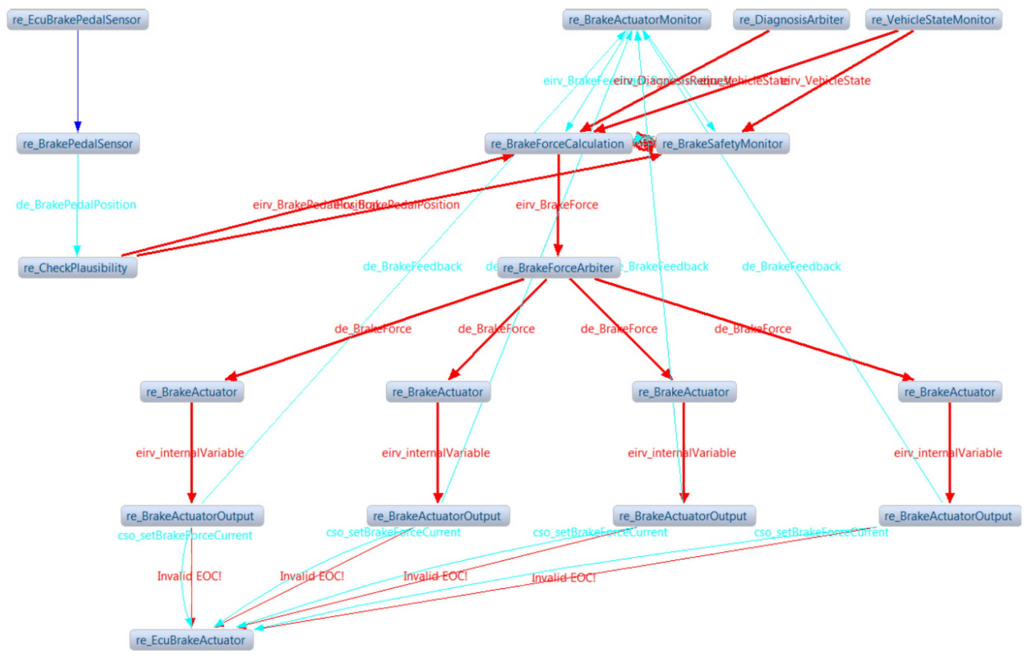
10.4.4. Evaluation of the Use Cases

The proposed method has been evaluated in the context of the breaking system example. The usage scenario follows the steps which are expected to be carried out by actual users of the dependency analysis tool. More specifically, the prototype which implements the developed methodology is employed to identify problems in the example model which hinder the derivation of parallelized schedulings. The semi-automatic fixes computed by the problem resolution algorithms are then applied and the changes are transferred back to the original model.

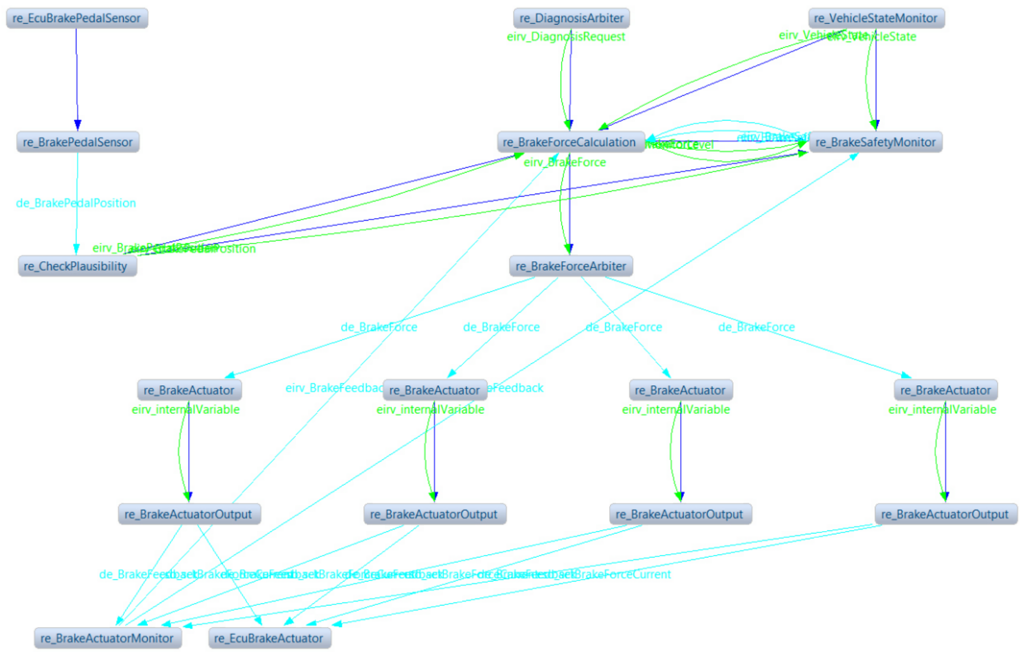
Figure 10.40(a) depicts a visualization of the results computed for the RunnableEntities of the breaking example model. A first glance already reveals that the analysis identified many problematic cases which must addressed in the next step. Typically, the first action a user is expected to take is the correction of errors for which the algorithm was able to determine automatic problem resolutions. In this case, the application of this function significantly reduces the number of problematic dependencies. However, the updated model still contains several cyclic dependencies which must be resolved manually. One example is the cycle which includes re_BrakeForceCalculation and re_BrakeSafetyMonitor. This situation can be corrected by creating a new AgeConstraint which specifically states that the entity may access data from a previous calculation cycle. This action can be justified by the assumption that monitoring is a continuous task and thereby using information from the last cycle is not a problem. In this case, the prototype is able to automatically generate a suitable AgeConstraint although the modification of the model must be approved by the developer. The next step addresses problems with the connections between re_BreakForceArbiter and the four instances of re_BrakeActuator. The analysis was not able to automatically resolve these incidences as the runnables possess different triggering frequencies which makes this type of problem resolution infeasible[37]. Resolving this problem requires the engineer to manually specify matching Age-

[36] http://www.eclipse.org/gef/zest/

[37] If triggering frequencies differ, it is possible that one runnable eventually overtakes another.



(a) TIMMO braking system after first analysis



(b) Braking system after using the analysis tool

Figure 10.40.: Application of the analysis to the breaking system model [Min12].

Constraints. Now, only the problematic connections between the four instances of re_BreakActuatorOutput and the re_EcuBrakeActuator remain. In this case, the analysis was able to determine that, while EOCs were specified, they are invalid because of the mismatching triggering frequencies. Since the constraints cannot be fulfilled, the engineer must either remove them or modify the system design.

After the application of these steps, the final state of the model, depicted in Figure 10.40(b), is free of errors. The model can now be used to map the runnables to tasks which can be executed in parallel on different processor cores while preserving a valid execution order. For this purpose, the authors of [Kie+14] describe how the analysis functions can be integrated into an overall development methodology which supports the parallelization of AUTOSAR-based systems.

In a domain where large models are very common, it is essential that the analysis can be completed in a feasible time frame. It is stated that the breaking example consists of 3000 lines of code[38] containing 200,000 characters and that the complete analysis (on an average notebook computer) takes about 2.4 seconds. The author concludes that the *"case study shows that the analysis tool is able to cope with medium sized systems and that changes suggested by the analysis can be easily performed"*. A recent case study carried out in the context of the ongoing WEMUCS project (`http://www.multicore-tools.de`) suggests that the analysis is also applicable to very large system models.

10.4.5. Summary and Discussion

This case study introduces a use case in the field of model-based development for embedded systems, more specifically the AUTOSAR domain. This consortium consists of a large number of car manufacturers, suppliers and software companies who currently face the challenge of migrating from single to multicore systems. It can therefore be asserted, that the techniques developed by [Kie+14; Min12] are very relevant to this industry.

The proposed analysis methodology enables the validation and semi-automatic refinement of automotive software designs. The methods provided by this approach are an important prerequisite for a subsequent parallelization of the modeled systems. For this purpose, the analysis identifies and classifies different types of data dependencies between RunnableEntities which represent the smallest functional units in AUTOSAR models. By detecting problematic backward dependencies between runnables and by offering semi-automatic problem resolution advisories, engineers are able to iteratively refine their development artifacts. The corrected and enriched models in turn represent the input for algorithms which derive optimized schedulings that map runnable sequences to processing cores. To evaluate the proposed solution, a prototype was implemented and the analysis functions were applied to detect problems in an existing model.

We will now summarize our findings:

Implementation effort

This case study has some unique characteristics. For example, the complex structure of the AUTOSAR language and the typically very large models necessitate a pre-analysis step in which the model is traversed and implicit information, such as the opposite ends of one-directional associations, are stored

[38]Stored in the AUTOSAR `arxml` file format.

in an object cache for quick access. This greatly reduces the effort for the execution of navigation operations during the analysis. The performance gain is especially relevant to the application of data-flow analysis since this method iteratively computes a large number of results. The computation of these helper maps can be easily integrated by encoding this action as a MAF evaluation macro which is executed as the first step of the evaluation strategy.

Another characteristic of this use case concerns the handling of concepts which are "instantiated" on the model level. More specifically, although each RunnableEntity conforms to a single model object, it may be referenced by different software components and thus possess multiple contexts with respect to the computation of DFA results. While the case study presented in Section 10.1 solves a similar problem by flattening the model's containment hierarchy, the use case described in this section employs a different approach. Instead of creating a copy of the respective element for each context, the data-flow rules operate on data structures which aggregate the results for all relevant contexts. For this purpose, the different "instances" of a runnable are encapsulated in EEI wrappers which uniquely identify each occurrence. This approach represents a valid solution with respect to DFA semantics although changes to a single instance result will lead to the recalculation of the complete result set. The prototypic implementation has shown that these features can be realized with little additional effort.

Classification

The analysis described in this case study represents one step of an analysis-assisted development methodology for automotive systems. It has been shown that the analysis approach can be integrated with existing methodologies in a complex application domain and is able to provide additional value to the developers. The analysis realizes several application scenarios: It enables the validation of AUTOSAR models and supports the enforcement of modeling guidelines by extracting information which is then used to derive problem resolution advisories for a tool-supported, semi-automatic refinement of the target models. The final result consists of an improved model which still conforms to the AUTOSAR standard and thus can be processed by any existing tool chain in this domain. Consequently, this case study demonstrates how the analysis technique can be employed to extend an existing development methodology.

Reuse

In a broader sense, the presented analyses reuse the idea of computing successor sets (cf. Section 10.1.2). However, the unique structure of AUTOSAR models and the aforementioned complications of having to deal with instantiation on the model level require a different approach to implement this method.

Usability

The authors of the proposed analysis approach have shown that it represents a feasible solution in the context of an industrial application scenario. The method was practically evaluated in the context of a prototypic implementation which provides support for semi-automatic problem resolution. It is also described how this technique can be integrated with automotive development processes, thereby improving existing methodologies and providing important groundwork for the migration to multicore systems.

Performance

The analysis has been applied to the breaking system example developed as part of the TIMMO project. It has been found that the analysis performs reasonably well for this medium-sized model and the use of the provided tools represents a beneficial addition to automotive development processes.

Several extensions points for future work have been identified which are likely to improve both the usefulness of the analysis approach and the integration with the AUTOSAR methodology. It is, for example, suggested that an initial set of AgeConstraints could be generated from Matlab models, thereby providing better input for the analysis [Min12]. It is also mentioned, that this technique would benefit from an in-depth analysis of the runnables' source code. This would make it possible to repartition existing code by determining the exact time when variables are written or read. As a consequence, depending on the respective properties of the runnable, it would be possible to derive parallelized schedules for entities with conflicting data dependencies. Analyzing the exact conditions in which multiple accesses to a variable may occur at the same time would provide the scheduler with additional opportunities for parallelization. This could be implemented in the form of adaptions of traditional DFA methods such as *reaching definitions*. [Kie+14] also proposes the use of SESE decomposition to cluster parallelizable regions in AUTOSAR models. The development process itself could be improved through a computation of metrics and an enforcement of modeling guidelines. [Min12] suggests that the *"number of unsolved backward dependencies can be used as a metric for the work that is necessary to make the new system run on multiple processing cores"* and that *"one can estimate the degree of freedom for mapping the software components and runnable entities to processing cores"*.

11. Conclusions and Outlook

In this thesis, we presented an approach that supports the specification and evaluation of static model analyses based on techniques originating from the area of compiler construction. This chapter contains concluding remarks concerning the overall methodology and discusses its relationship with the initially stated goals and lists possible extensions of the developed approach.

Section 11.1 summarizes the initial research questions, the methods which were chosen to address these problems and provides an overview of the contributions of this thesis. Section 11.2 discusses how these results relate to the problems and challenges that were listed in the beginning. Finally, Section 11.3 investigates possible extension points for the proposed approach and identifies additional research questions which can be addressed in future work on this topic.

11.1. Summary

The development of a data-flow based approach to model analysis was motivated by the fact that contemporary techniques such as OCL are limited in their expressiveness which complicates their application in certain use cases. With the rising importance and the more wide-spread use of modeling technologies, especially in the context of the recent trend towards domain-specific languages, the need for an analysis method that supports an intuitive and concise specification and evaluation of semantic properties has increased significantly. The fact that the underlying principles of the technological space of compiler construction bear a strong resemblance to the language stack that constitutes the foundation of modeling gave rise to the idea of adapting existing and well-established static analysis methods for semantic validation for use in the modeling domain. For this purpose, we studied the techniques of attribute grammars and data-flow analysis, two methods that are standard practice in the compiler construction domain which and have been successfully employed for some decades. While attribute grammars support the enrichment of a target language's definition with declarative analysis specifications, the data-flow analysis method supports a context-sensitive evaluation of language instances and applies fixed-point semantics to approximate runtime behavior. Based on a comparison of the properties of both technological spaces, we derived a suitable architecture which implements an adaption of the flow analysis concept for the modeling domain and detailed the realization of corresponding methods for the definition and execution of flow-based analyses on models.

Goals

In Chapter 1, we detailed our objectives along with a set of requirements which have to be fulfilled by any method which intends to provide a viable solution to the presented challenges. Since modeling technologies are usually employed during the design phase of development processes, it is essential for a suitable analysis technique to support a static approximation of the behavior of the modeled systems to enable an early detection of errors. We surmised that, as an extension of this objective, the approach has to implement a unified method for the realization of different usage scenarios such as validation tasks, the computation of model metrics or the assessment of a model's quality. It was also stated that the extraction of useful semantic properties requires the consideration of the overall context of language elements. Additionally, we emphasized that, on the technical level, a consistent implementation of any analysis method intended for use in the modeling domain should itself also be based on modeling technology and it has to define appropriate connection points to the relevant modeling standards. Furthermore, it was argued that the design must support non-intrusive specifications, so as not to require any modification to existing standards, tooling and modeling artifacts. On a conceptual level, it is important that the two different usage scenarios of analysis specification and analysis execution are considered since they are usually carried out by different users with diverging interests. Finally, we stated the importance of a practical evaluation of the viability of the proposed method in the context of a reference implementation and the realization of multiple case studies.

Method

The process of developing and implementing a suitable approach for static model analysis involved the following steps:

1. In Chapter 2, we introduced the underlying concepts and technologies of the relevant technological spaces. In this context, we provided an overview of the compiler construction domain and presented the analysis techniques of attribute grammars and data-flow analysis in greater detail. Additionally, we gave a summary of the basic concepts and principles of the technological space of modeling with a focus on the notion of abstraction layers and the static semantics of modeling languages. In Chapter 3, we presented contemporary techniques for static (model) analysis and discussed their relationships with our own approach.

2. Since the proposed method for model analysis is based on a combination of techniques originating from the compiler construction domain and their adaption to the field of modeling, we based the design of our approach on an in-depth study of the similarities and differences of both technological spaces. For this purpose, we examined the relevant properties of both domains and subsequently derived the design goals for our approach in Chapter 4. To support the integration of the analysis technology with the standards and prac-

tices of the MDE domain, we put a special emphasis on the connection of the analysis artifacts with modeling constructs and described how domain-specific data-flow paths in models can be implemented using a dependency discovery mechanism. Based on these results, we provided a formal as well as a technical specification of the proposed techniques in Chapter 5 and Chapter 6. For this purpose, we defined a model-based DSL for analysis specifications, supplied the necessary instantiation semantics and adapted the traditional algorithms for evaluating DFA equation systems to incorporate the facilities required for handling dynamic dependency relationships and demand-driven evaluation.

3. Chapter 7 and Chapter 8 described the reference implementation of our approach, the Model Analysis Framework. In these chapters we presented the basic architecture and the relevant components of the Eclipse RCP platform which was chosen as technical foundation for our tooling since it provides an extensible and highly customizable Open Source development and runtime framework for model-based technologies. To motivate the architectural design of MAF, we listed a set of design goals to ensure that the reference implementation constitutes a flexible tooling environment that is able to function both as a research platform and as an environment that is suitable for use in industrial application scenarios. We then detailed the MAF Core component which provides a range of in-built functionality such as a central repository management system and notification services as well as interfaces for exchangeable modules including resource adapters for specific input artifact types, rule implementation languages and different DFA solving strategies. Additionally, we described the Analysis Editor component and the Project Set concept which together realize an IDE for analysis development and testing.

4. The practical application of the proposed methods has been evaluated in the context of several common application scenarios and case studies. In Chapter 9, we listed templates for often used analyses which can be easily adapted to different domains. This standard library of analysis functions provides a basis for the implementation of custom analyses as, with the modular analysis specification approach, it is possible to leverage existing analyses by either modifying them or by accessing their precomputed results in other specifications. This concept has been illustrated in the context of the case studies described in Chapter 10 where we demonstrated how the standard analyses can be adapted to work in different domains and can be combined to implement more complex functionality. In the case studies we focused not only on applying the analysis method to different domains but also on addressing a range of different application scenarios such as the validation of models, the computation of metrics and the extraction of semantic properties.

Contributions

The development of the analysis technique along the lines of the initially stated objectives resulted in a set of theoretical and practical contributions. These include a study of the similarities and differences between the technological spaces of modelware and grammarware with an emphasis on the alignment and usage of abstraction layers in both areas and considerations regarding different aspects of the role played by the abstract and concrete syntax and the static semantics of formal and model-based languages. Along with the derived challenges and design goals that influenced the development of our own approach, these results may also be of interest when addressing related problems in the area of Software Language Engineering.

Contributions relating to the flow-based analysis approach itself consist of mathematical and technical specifications of the language constructs that constitute flow-based analyses and their relationships with modeling artifacts as well as the corresponding semantics that govern their instantiation and execution. More specifically, these results include model-based domain-specific languages for analysis specification, instantiation and configuration. Furthermore, traditional DFA algorithms were adapted to support a demand-driven fixed-point evaluation of data-flow equation systems relying on the dependency discovery mechanism for handling data-flow paths which are implicitly contained in the rule implementations. While these concepts were specifically designed for use in the modeling domain, it is conceivable that they can also be ported back to the original application field of compiler construction.

On the practical side, with the Model Analysis Framework, we supplied a reference implementation which serves both as a fully featured IDE for the implementation of model analyses and as a light-weight functional runtime library. The modular approach and the flexible architecture of MAF enables its usage as a research platform, e.g. to test out new solving strategies, while at the same time providing a reliable, high-quality environment supporting the extension of existing third-party tools with analysis functionality. Finally, a standard library containing a set of analysis templates was defined and several case studies were carried out which not only served to validate the practical applicability of the approach but can also be used as starting points for custom implementations.

11.2. Discussion

In Section 1.1 we identified the problems that exist in the context of the application of analysis technologies in the modeling domain and derived a set of challenges that must be addressed in order to remedy this situation. Based on these challenges, Section 1.2 lists the concrete objectives and details an approach which enables the implementation of a suitable solution. We will now discuss how the methods developed in this thesis relate to these issues.

Challenge 1: Test the models, not the system

Since modeling is often used during the early phases of development processes, errors in models may have a significant impact on later stages. While it is often easy to detect syntactical errors early on, the evaluation of semantic properties is usually a more difficult task. In Challenge ①, we therefore stressed the importance of providing a powerful static analysis technique for the modeling domain. The flow-based approach to model analysis realizes this requirement by offering the possibility to enrich modeling languages with specifications of their static semantics. In the case studies, we have shown how this concept can be used to, for example, validate the structural integrity of business processes (cf. Section 10.1.2) and model-based representations of natural language information (cf. Section 10.3).

Challenge 2: Approximate Dynamic Behavior

Challenge ② further emphasized the focus on static analysis, i.e. the evaluation of the language instance itself, to derive useful information about a system's runtime properties. It was mentioned that, since modeling languages are usually tailored to a specific application domain, the analysis technique has to be able to incorporate domain-specific information. With the presented technique, this is possible due to a declarative, attribute-based approach to analysis specification which supports the enrichment of the target language with analysis constructs. This feature also bridges the technological gap which exists between the definitions of the analyses and their target language artifacts. Since data-flow analysis relies on the propagation of values, this method incorporates flow-sensitive information in the evaluation process and by using fixed-point semantics for solving the DFA equation systems, the dynamic behavior of a modeled system can be approximated. Overall, this approach can therefore be considered to be a suitable solution to the presented challenge.

Challenge 3: Support Domain-independence and Reusability

In the third challenge, we further stressed the importance of a unified approach which is able to support different application scenarios while, at the same time, being generic in the sense that it does not presume the existence of domain-specific features. Consequently, our method relies on a very limited set of basic assumptions that must be met by the target modeling framework. More specifically, the analysis approach only presumes the existence of the instantiable class concept as a basis for the annotation and instantiation of analysis constructs. Additionally, because of their prevalence in many MDE frameworks and their effect on the availability of data-flow attributes at model elements, we also included the necessary semantics for the support of generalization relationships.

The versatility with respect to different application scenarios was evaluated in the context of several case studies. In addition to common validation tasks, we were also able to demonstrate how advanced analyses can be used to statically derive approximations of complex structural and dynamic properties. For example, in Section 10.1.3, data-flow analysis was used to examine the availability of resource objects in business process instances while Section 10.4 computed information for the iterative refinement of AUTOSAR models. An example for an analysis which focuses on structural properties has been provided in Section 10.1.4 which implemented an algorithm for the SESE decomposition of control-flow models.

In Challenge ③, we also addressed the different roles of language developers and users and their respective preferences with relation to the application of analysis technology in the modeling context. The textual analysis specification DSL with its library concept enables developers to efficiently create and manage analysis artifacts during the whole development cycle. We have also shown that the technique can be applied without requiring any modifications to existing practices and tooling. Similarly, from a user's perspective, the approach supports a seamless integration with productive environments as demonstrated, for example, in Section 10.1.6.

Challenge 4: Integrate with Modeling Practices

Challenge ④ demanded that the developed technology has to integrate with existing standards and practices in the modeling domain. This issue has been addressed by comparing the technological spaces of modelware and grammarware and subsequently identifying design goals which enable a consistent integration of the DFA technique with modeling standards. Notably, the main artifacts of the analysis method are themselves based on MDE technology. For example, the specification language has been defined as a metamodel while the concrete syntax was realized using the SLE-centric Xtext framework which connects context-free grammars to modeling constructs. Furthermore, as mentioned in the previous item, support for basic modeling concepts such as classes and generalizations is an integral part of the approach. Adherence to existing standards is ensured through an extension mechanism which enriches MOF's meta language layer *M3* with definitions for instantiable analysis artifacts without requiring any modification of MOF itself (cf. Section 6.1.1). This non-intrusive design also applies to the tasks of analysis specification and execution. Neither the target metamodel on *M2* nor the respective *M1* models have to be modified in any way to support the definition and the evaluation of analyses.

To ensure a consistent integration with the modeling domain, further adaptions of the data-flow analysis method were devised and implemented: Since model graphs possess no inherent flow direction which is independent of domain-specific semantics, the need for declaring a flow direction as commonly required in DFA frameworks (or the inheritance/synthesis direction of attribute grammars respectively) was eliminated. Instead, we rely on a demand-driven approach in which data-flow rules are able to request required input values. For this purpose, we adapted the traditional algorithms for solving flow equation systems so that they are able to dynamically construct a representation of the dependency relationships between attribute instances and schedule a valid execution order accordingly. It has been shown that this feature can be implemented in a performant fashion. A positive side-effect of this approach can be found in the fact that the dependency graph overlays the model graph. This means that data-flow values may be propagated between arbitrary attribute instances and do not have to follow the layout imposed by the model's edges. This is important since the distinction between what constitutes a node and which concept represents an edge is domain-specific information and therefore has to be implemented by the analysis itself. The presented approach inherently supports this requirement. Because of the versatile nature of analyses in the modeling domain, we decided to lift the requirement of using formal value domains (semilattices) in favor of freely configurable datatypes. Since the usage of arbitrary value domains may complicate the specification of an element $\top$ which is neutral with respect to the application of the confluence operator, we instead introduced a unique constant INIT which can be used by implementations of data-flow rules to correctly handle cyclic dependency relationships.

11.3. Outlook

The descriptions that have been provided in this thesis cover the basic motivation for the proposed approach, its underlying principles as well as formal and technical specifications of the components which constitute the analysis method. Furthermore, a reference architecture was developed and the technique has been evaluated in the context of a variety of different scenarios and problem domains. In this sense, this work provides detailed descriptions of all necessary aspects required for a successful application of the flow-based analysis technology.

Nevertheless, going beyond the requirements of the initially stated objectives, there is some potential for implementing extensions of the devised methodologies both on a theoretical and on a practical level. Consequently, in the following, we will discuss several topics which - in our opinion - warrant future research work. Some of these items expand on the concepts and definitions that represent the foundation of the approach while others introduce new aspects that are aimed at enabling a more focused application of the developed techniques.

Extended Analysis Library Concept

In Section 6.2, we motivated the choice of a textual format for the specification of attributions. In this context, we also mentioned that this concept can be extended to include support for complex, interconnected libraries in the notion of class libraries as found in object-oriented programming languages. While the current approach is completely sufficient for the specification of any analysis that conforms to the prerequisites of flow-based fixed-point evaluations, the implementation of this feature can still be beneficial as it would streamline the specification process. If users were able to organize whole libraries of interdependent analysis definitions, this would improve the aspects of reuse and maintainability by further reducing redundancy in the process of analysis development. This is especially true if analysis functions are reused across different projects and are worked on by multiple developers.

An implementation of these features would require the extension of the analysis specification metamodel (AttrMM) and the corresponding textual domain-specific language with constructs that enable the declaration of inheritance relationships between attributions. For this purpose, an attribution must be able to declare extension relationships to one or more other attributions. In this case, the specialized attribution would inherit attribute definitions, occurrences and semantic rules from its parent(s). To support polymorphism, it must be possible to overwrite inherited constructs in extending attributions if they possess the same identifier. This behavior can be further customized through the implementation of visibility modifiers such as public or private and final. The actual processing of the inheritance relationships can be implemented statically in a similar fashion to the enhancement of attributions with generalization relationships as described in Section 6.3.3. Alternatively, the instantiator and the rule invoker could be set up to dynamically select the correct version of a polymorph analysis construct at runtime.

Instantiable Analysis Templates

The adaption of existing analysis templates (such as the ones presented in Chapter 9) can be further simplified if analysis specifications were provided with the ability to define placeholders for metamodel bindings which can then be instantiated for specific application domains. Currently, a template must be adapted to the target metamodel by replacing the class references of attribute extensions. For example, in the case of JWT (cf. Section 10.1), the Node and Edge concepts of the reachability template must be substituted with the respective types ActivityNode and ActivityEdge. Furthermore, the navigational statements inside the data-flow rules must be adapted to retrieve their values using the correct model references. Complementing the analysis library mechanism with a placeholder concept to support the instantiation of analysis templates for different application domains would thereby further improve the maintainability and versatility of analyses.

To implement this functionality, a placeholder concept must be implemented for both the attribution and the rule specifications. A concrete realization of an analysis template can then implement the mappings between the respective class and reference types in the target metamodel and the placeholder in the analysis template. Just as in the previous item, this process would require an adaption of the AttrMM language to support the declaration of placeholders for metamodel classes and the navigable model references which are used inside the data-flow rules. Navigational statements inside the data-flow rules could, for example, be dynamically configured by passing the respective mappings to the rule's implementation alongside the execution context (cf. Section 6.4.1).

Graphical Syntax

As mentioned in Section 6.2, the usage of a text-based format for analysis specification has several advantages: It makes it simpler to manage analysis artifacts independently of models and does not require any modification of existing tooling ecosystems. Also, the specific properties of the attribute-based declarations, namely the usage of global attribute, datatype and rule definitions which can be referenced in different contexts, are easier to manage when using a textual format.

However, this does not imply that the implementation of a graphical syntax would not be possible or that there don't exist any circumstances in which a visual representation may actually be preferable. For example, a graphical syntax could provide a more intuitive way of communicating the structural properties of analyses when multiple developers are involved and, in this context, also serve as a form of documentation. This is especially true if the representation consists of a combination of the target metamodel and the attribute annotations in a similar way OCL constraints can be visualized as annotations at classes. It should also be noted that textual and graphical methods of displaying analysis artifacts are not mutually exclusive. If both methods were available, the user would be able to switch between both formats depending on the requirements of the current situation.

To implement this functionality, it would be necessary to first devise a suitable graphical syntax for analyses and to specify appropriate mappings to the constructs defined in the abstract syntax. This can, for example, be accomplished with editor generators for model-based DSLs such as the Eclipse Graphical Modeling Framework (GMF). Alternatively, existing tooling environments could be extended with the required functionality. For example, the graphical syntax for attributions can be integrated with EMF's diagram-based metamodel editor to support the simultaneous development and management of model and analysis artifacts. In any case, the developed tooling must address the inherent problem of global specifications, for example by providing dedicated dialogs which support the management of global attribute, datatype and rule definitions.

Additional Application Domains

In Chapter 10, we studied the application of the analysis technique in the context of a range of different application domains and different usage scenarios. To further examine the applicability of the approach and to build a more comprehensive library of analysis functions, it would be beneficial to extend this study to additional areas.

This could include the evaluation of new application domains as well as a focus on different usage scenarios. For example, it would be possible to realize an implementation of modeling metrics and guidelines for the UML [GPC05; Ber04] and BPMN [GL06] or other languages. Furthermore, the analysis of alternative paths in control-flow models (cf. Section 9.2.4) can be extended to derive test paths for model-based testing approaches according to different coverage criteria [UL07]. For a detailed study of the relationship between the model-based analysis technique and the traditional DFA approach in the domain of compiler construction, the developed technique could be integrated with the GeCoS compiler suite [Der+] which transforms program code into a model-based representation.

Anticipation of Changes to Modeling Languages

The results of a data-flow analysis are always computed locally in the context of a specific attribute instance and subsequently propagated throughout the model. Typically, the input parameters for the computation of a local data-flow result are acquired from model properties and from data-flow attributes located at adjacent objects. As a consequence, the navigational expressions used in the data-flow rules are often very simplistic in nature as they are commonly used to address elements which are immediate neighbors of the context object. This becomes especially evident when compared to the more complex navigation statements of OCL constraints which encode the full path to the target object or property. It can therefore be assumed that DFA-based definitions provide a higher level of invariance against structural changes in the underlying modeling language.

This concept was partially explored in the EAM case study (cf. Section 10.2), in which changes to the metamodel were anticipated through a generic and unified representation of metamodel and model data. The respective analyses were defined in a way that enables them to carry out the computations independently of the concrete structure of the underlying modeling language. In future work, the possibilities offered by this approach could be studied in greater detail. With relation to the Object Constraint Language, the concrete circumstances and the extent to which these statements could be replaced by more flexible data-flow based definitions could be examined, possibly resulting in a methodology for translating existing OCL constraints into data-flow based specifications.

Development Methodology

This thesis details the conceptual and the technical aspects of the flow-based analysis approach. To enable a consistent integration of the devised techniques with existing development processes, the methodical properties of the application of analysis functionality have to be studied as well.

For this purpose, it would certainly be beneficial to examine the usage of the analysis methods in practical scenarios. This could, for example, be accomplished by monitoring a set developers and evaluating how they employ the provided functions. Based on these observations, potential challenges and problems could be identified and the approach could be refined with respect to its usability. The results can, for example, be used to improve different aspects of the analysis concept such as the syntax of the analysis specification language or the management of model and analysis artifacts. Furthermore, a set of guidelines and best practices can be devised aimed at supporting developers in the utilization of the provided functions. These concepts can then also be adapted to the specific requirements of existing model-based development approaches such as the Model-driven Architecture or the Rational Unified Process to support a fully integrated development methodology.

Part V.

Annex

Bibliography

[BPMN] *Business Process Modeling Language (BPMN) 2.0 Specification*. Object Management Group, Jan. 2011. URL: http://www.omg.org/spec/BPMN/2.0/.

[EMFP] *Eclipse Modeling Framework (EMF) Project Page*. URL: http://eclipse.org/modeling/emf/.

[MDA] *Model-Driven Architecture Guide 1.0.1*. Object Management Group, 2003. URL: http://www.omg.org/mda/.

[MOF] *Meta Object Facility Core 2.4.1 Specification*. Aug. 2011. URL: http://www.omg.org/spec/MOF/2.4.1/.

[OCL] *Object Constraint Language (OCL) 2.3.1 Specification*. Object Management Group, Jan. 2012. URL: http://www.omg.org/spec/OCL/2.3.1/.

[OSGi] *OSGi Service Platform Core Specification, Release 4.1*. http://www.osgi.org/Specifications. 2007.

[QVT] *Query / View / Transformation 1.1 Specification*. Object Management Group, Jan. 2011. URL: http://www.omg.org/spec/QVT/1.1/.

[TMF] *Eclipse Textual Modeling Framework (TMF)*. URL: http://www.eclipse.org/modeling/tmf/.

[UML] *Unified Modeling Language 2.4.1 Specification*. 2011. URL: http://www.omg.org/spec/UML/2.4.1/.

[UMLi] *Unified Modeling Language 2.4.1 Infrastructure Specification*. May 2011. URL: http://www.omg.org/spec/UML/2.4.1/.

[UMLs] *Unified Modeling Language 2.4.1 Superstructure Specification*. June 2011. URL: http://www.omg.org/spec/UML/2.4.1/.

[XMI] *XML Metadata Interchange (XMI) 2.4.1 Specification*. Object Management Group, Aug. 2011. URL: http://www.omg.org/spec/XMI/2.4.1.

[XTEX] *Xtext - Language Development Framework*. URL: http://www.eclipse.org/Xtext/.

[Aal07] W.M.P. van der Aalst. "Trends in Business Process Analysis: From Verification to Process Mining". In: *Proceedings of the 9th International Conference on Enterprise Information Systems*. ICEIS'07. INSTICC, 2007, pp. 12–22.

[Aal98] W.M.P. van der Aalst. "The Application of Petri nets to Workflow Management". In: *Journal of circuits, systems, and computers* 8.01 (1998), pp. 21–66.

[ABK07] Kyriakos Anastasakis, Behzad Bordbar, and Jochen M Küster. "Analysis of Model Transformations via Alloy". In: *Proceedings of the 4th Workshop on Model-Driven Engineering, Verification and Validation.* MoDeVVa'07. 2007, pp. 47–56.

[AC76] F.E. Allen and J. Cocke. "A Program Data flow Analysis Procedure". In: *Communications of the ACM* 19.3 (1976), p. 137.

[AD97] L. Apfelbaum and J. Doyle. "Model based Testing". In: *Software Quality Week Conference.* 1997, pp. 296–300.

[ADS12] Manar H Alalfi, James R Cordy Thomas R Dean, and Matthew Stephan Andrew Stevenson. "Models are Code too: Near-miss Clone Detection for Simulink Models". In: *ICSM.* Vol. 12. IEEE, 2012, pp. 295–304.

[Aho+06] Alfred V. Aho et al. *Compilers: Principles, Techniques, and Tools.* 2nd. Addison Wesley, 2006. isbn: 0321486811.

[AK03] Colin Atkinson and Thomas Kuhne. "Model-driven Development: A Metamodeling Foundation". In: *IEEE Software* 20.5 (2003), pp. 36–41.

[All70] Frances E. Allen. "Control flow Analysis". In: *SIGPLAN Not.* 5.7 (1970), pp. 1–19.

[Ana+07] Kyriakos Anastasakis et al. "UML2Alloy: A Challenging Model Transformation". In: *Proceedings of the 10th International Conference on Model Driven Engineering Languages and Systems.* MoDELS'07. Springer-Verlag, 2007, pp. 436–450.

[Ana+10] K. Anastasakis et al. "On challenges of Model Transformation from UML to Alloy". In: *Software and Systems Modeling* 9.1 (2010), pp. 69–86.

[AP04] Marcus Alanen and Ivan Porres. *A Relation between Context-Free Grammars and Meta Object Facility Metamodels.* Tech. rep. TUCS, 2004.

[Ass11] E.G. Assembly. "Ecmascript language specification - Version 5.1". In: *Standard ECMA-262* (2011).

[B+05] J. Bézivin, I. Kurtev, et al. "Model-based Technology Integration with the Technical Space Concept". In: *Proceedings of the Metainformatics Symposium.* Springer-Verlag, 2005.

[Baa03] Thomas Baar. "The Definition of Transitive Closure with OCL - Limitations and Applications". In: *Ershov Memorial Conference.* Vol. 2890. Springer-Verlag, 2003, pp. 358–365.

[Bac+57] J.W. Backus et al. "The FORTRAN Automatic Coding System". In: *Papers presented at the February 26-28, 1957, Western Joint Computer Conference: Techniques for reliability.* ACM. 1957, pp. 188–198.

[Bas+09] Muthu Manikandan Baskaran et al. “Compiler-assisted dynamic Scheduling for effective Parallelization of Loop Nests on Multicore Processors”. In: *ACM Sigplan Notices*. Vol. 44. 4. ACM. 2009, pp. 219–228.

[BDW06] Achim D. Brucker, Jürgen Doser, and Burkhart Wolff. “Semantic Issues of OCL: Past, Present, and Future”. In: *Electronic Communications of the EASST* 5 (2006).

[Beh+10] Heiko Behrens et al. *Xtext Documentation*. 2010. URL: http://www.eclipse.org/Xtext/documentation/1_0_1/xtext.pdf.

[Ber04] Brian Berenbach. “The Evaluation of large, complex UML Analysis and Design Models”. In: *Proceedings of the 26th International Conference on Software Engineering*. ICSE’04. IEEE. 2004, pp. 232–241.

[Bez05] J. Bézivin. “On the Unification Power of Models”. In: *Software and Systems Modeling* 4.2 (2005), pp. 171–188.

[BLL10] Lionel Briand, Yvan Labiche, and Q Lin. “Improving the Coverage Criteria of UML State Machines using Data flow Analysis”. In: *Software Testing, Verification and Reliability* 20.3 (2010), pp. 177–207.

[BMS09] S. Buckl, F. Matthes, and C. M Schweda. “Classifying Enterprise Architecture Analysis Approaches”. In: *Enterprise Interoperability* (2009), pp. 66–79.

[Boe+05a] F.S. de Boer et al. “Change Impact Analysis of Enterprise Architectures”. In: *Proceedings of the 6th International Conference on Information Reuse and Integration*. IRI’05. 2005, pp. 177–181.

[Boe+05b] F.S. de Boer et al. “Enterprise Architecture Analysis with XML”. In: *Proceedings of the 38th Annual Hawaii International Conference on System Sciences*. HICSS’05. IEEE, 2005.

[Boh02] Shawn A Bohner. “Software Change Impacts - An evolving Perspective”. In: *Proceedings of the 10th International Conference on Software Maintenance*. ICSM’02. IEEE. 2002, pp. 263–272.

[Bra09] M. van den Brand. “Model-driven Engineering meets generic Language Technology”. In: *Software Language Engineering* (2009), pp. 8–15.

[Bur+11] C. Bürger et al. “Reference Attribute Grammars for Metamodel Semantics”. In: *Software Language Engineering* (2011), pp. 22–41.

[Bus+11] Markus Buschle et al. “A Tool for Enterprise Architecture Analysis using the PRM Formalism”. In: *Information Systems Evolution*. Springer-Verlag, 2011, pp. 108–121.

[Bus+96] Frank Buschmann et al. *A System of Patterns: Pattern-oriented Software Architecture*. 1996.

[BW02] Achim D. Brucker and Burkhart Wolff. “A Proposal for a Formal OCL Semantics in Isabelle/HOL”. In: *Theorem Proving in Higher Order Logics*. Vol. 2410. TPHOLs’02. Springer-Verlag, 2002, pp. 99–114.

[CC04] W. Croft and D.A. Cruse. *Cognitive Linguistics*. Cambridge Textbooks in Linguistics. Cambridge University Press, 2004. isbn: 978-0521667708.

[CCR08] Jordi Cabot, Robert Clarisó, and Daniel Riera. “Verification of UML/OCL Class Diagrams using Constraint Programming”. In: *Proceedings of the International Conference on Software Testing Verification and Validation Workshop*. ICSTW’08. IEEE, 2008, pp. 73–80.

[CHK01] Keith D Cooper, Timothy J Harvey, and Ken Kennedy. “A simple, fast dominance Algorithm”. In: *Software Practice & Experience* 4 (2001), pp. 1–10.

[Cho56] N. Chomsky. “Three Models for the Description of Language”. In: *IRE Transactions on Information Theory* 2.3 (1956), pp. 113–124.

[CK01] María Victoria Cengarle and Alexander Knapp. “A Formal Semantics for OCL 1.4”. In: *Proceedings of the 4th International Conference on the Unified Modeling Language*. Vol. 2185. UML’01. Springer-Verlag, 2001, pp. 118–133.

[CK04a] María Victoria Cengarle and Alexander Knapp. “OCL 1.4/5 vs. 2.0 Expressions Formal Semantics and Expressiveness”. In: *Software and Systems Modeling* 3.1 (2004), pp. 9–30.

[CK04b] M.V. Cengarle and A. Knapp. “UML 2.0 Interactions: Semantics and Refinement”. In: *Proceedings of the 3rd Int. Wsh. Critical Systems Development with UML*. CSDUML’04. 2004, pp. 85–99.

[Coc70] J. Cocke. “Global Common Subexpression Elimination”. In: *Proceedings of the Symposium on Compiler Optimization*. ACM. 1970, pp. 20–24.

[CR08] Eric Clayberg and Dan Rubel. *Eclipse Plug-ins*. 3rd ed. Addison-Wesley, 2008. isbn: 978-0321553461.

[Dei+08] Florian Deissenboeck et al. “Clone Detection in Automotive Model-Based Development”. In: *Proceedings of the 30th International Conference on Software Engineering*. ICSE’08. IEEE. 2008, pp. 603–612.

[Den13] Hadi Deniz. “Konzeption und Implementierung eines DSL Editors für modellbasierte Datenflussanalyse”. MA thesis. University of Augsburg, 2013.

[Der+] Steven Derrien et al. *GeCoS: Generic compiler suite*. url: http://gecos.gforge.inria.fr/.

[DKV00] Arie van Deursen, Paul Klint, and Joost Visser. “Domain-specific Languages: An annotated Bibliography”. In: *ACM SIGPLAN Notices* 35.6 (2000), pp. 26–36.

[DM98] Leonardo Dagum and Ramesh Menon. “OpenMP: An Industry Standard API for shared-memory Programming”. In: *Computational Science & Engineering, IEEE* 5.1 (1998), pp. 46–55.

[EH07] Torbjörn Ekman and Görel Hedin. “The JastAdd System - Modular extensible Compiler Construction”. In: *Science of Computer Programming* 69.1 (2007), pp. 14–26.

[EK99] Andy Evans and Stuart Kent. “Core Meta-modelling Semantics of UML: The pUML Approach”. In: *UML'99 - The Unified Modeling Language* (1999), pp. 754–754.

[ET12] D.W. Embley and B. Thalheim. *Handbook of Conceptual Modeling: Theory, Practice, and Research Challenges*. Springer-Verlag, 2012. isbn: 978-3642158650.

[EV06] S. Efftinge and M. Völter. “oAW xText: A framework for textual DSLs”. In: *Workshop on Modeling Symposium at Eclipse Summit*. Vol. 32. 2006.

[FB10] Wolf Fischer and Bernhard Bauer. “Combining Ontologies And Natural Language”. In: *Advances in Ontologies* (2010), p. 27.

[FB11] Wolf Fischer and Bernhard Bauer. “Cognitive-linguistics-based Request Answer System”. In: *Proceedings of the 7th International Conference on Adaptive Multimedia Retrieval: Understanding Media and adapting to the User*. AMR'09. Springer-Verlag, 2011.

[Fen+06] Helmut Fennel et al. “Achievements and Exploitation of the AUTOSAR Development Partnership”. In: *SAE Convergence Congress* 2006 (2006), p. 10.

[FFJ09] Ulrik Franke, Waldo Rocha Flores, and Pontus Johnson. “Enterprise Architecture Dependency Analysis using Fault Trees and Bayesian Networks”. In: *Proceedings of the Spring Simulation Multiconference*. SpringSim'09. Society for Computer Simulation, 2009, p. 55.

[FH02] Peter H. Feiler and Watts S. Humphrey. “Software Process Development and Enactment: Concepts and Definitions”. In: *ICSP*. Jan. 3, 2002, pp. 28–40.

[Fis13] Wolf Fischer. “Linguistically Motivated Ontology-Based Information Retrieval”. PhD thesis. University of Augsburg, 2013.

[Gam+95] E. Gamma et al. *Design patterns: Elements of reusable Object-oriented Software*. Vol. 206. Addison-Wesley, 1995.

[Gar08] Luciano García-Bañuelos. “Pattern Identification and Classification in the Translation from BPMN to BPEL”. In: *Proceedings of the Confederated International Conferences, CoopIS, DOA, GADA, IS, and ODBASE 2008*. Vol. 5331. Springer-Verlag, 2008, pp. 436–444.

[GBL05] Vahid Garousi, Lionel C Briand, and Yvan Labiche. “Control Flow Analysis of UML 2.0 Sequence Diagrams”. In: *Model Driven Architecture–Foundations and Applications*. Springer-Verlag, 2005, pp. 160–174.

[GBR03] Martin Gogolla, Jörn Bohling, and Mark Richters. “Validation of UML and OCL Models by Automatic Snapshot Generation”. In: *Proceedings of the 6th International Conference on the Unified Modeling Language*. UML’03. Springer-Verlag, 2003, pp. 265–279.

[GBR07] Martin Gogolla, Fabian Büttner, and Mark Richters. “USE: A UML-based Specification Environment for validating UML and OCL”. In: *Science of Computer Programming* 69.1 (2007), pp. 27–34.

[Gee05] D. Geer. “Eclipse becomes the dominant Java IDE”. In: *IEEE Computer* 38.7 (2005), pp. 16–18.

[Geo+04] Loukas Georgiadis et al. “Finding Dominators in Practice”. In: *Proceedings of the 12th European Symposium on Algorithms*. ESA’04. Springer-Verlag, 2004, pp. 677–688.

[Gho10] Debasish Ghosh. *DSLs in Action*. Manning Publications, 2010. isbn: 1935182455.

[GL06] Volker Gruhn and Ralf Laue. “Complexity Metrics for Business Process Models”. In: *Proceedings of the 9th International Conference on Business Information Systems*. Vol. 10. BIS’06. 2006, pp. 1–12.

[Gog07] Martin Gogolla. “Model Development in the UML-based Specification Environment (USE)”. In: *Methods for Modelling Software Systems (MMOSS)*. Dagstuhl Seminar Proceedings 06351. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2007.

[Got+09] Mathias Götz et al. “Token Analysis of Graph-Oriented Process Models”. In: *Proceedings of the 2nd International Workshop on Dynamic and Declarative Business Processes in the context of the 13th IEEE International EDOC Conference*. DDBP’09. 2009.

[GPC05] Marcela Genero, Mario Piattini, and Coral Calero. “A Survey of Metrics for UML Class Diagrams”. In: *Journal of Object Technology* 4.9 (2005), pp. 59–92.

[Gru+00] Dick Grune et al. *Modern Compiler Design*. 1st. John Wiley & Sons, Inc., 2000. isbn: 0471976970.

[GS04] Brian J. Gough and Richard M. Stallman. *An Introduction to GCC*. Network Theory Ltd., 2004.

[Hae+12] Bernhard Haeupler et al. “Incremental Cycle Detection, Topological Ordering, and Strong Component Maintenance”. In: *ACM Trans. Algorithms* 8.1 (2012), 3:1–3:33.

[Hec77] M.S. Hecht. *Flow Analysis of Computer Programs*. Elsevier Science Inc., 1977.

[Hed00] Görel Hedin. “Reference Attributed Grammars”. In: *Informatica (Slovenia)* 24.3 (2000).

[HMU79] J.E. Hopcroft, R. Motwani, and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Vol. 3. Addison-Wesley, 1979. isbn: 0321455363.

[HP12] V. Haren and Van Haren Publishing. *ArchiMate 2. 0 Specification.* The Open Group. Van Haren Publishing, 2012. isbn: 978-9087536923.

[HU69] John E. Hopcroft and Jeffrey D. Ullman. *Formal Languages and their Relation to Automata.* Boston, MA, USA: Addison-Wesley, 1969.

[HU72] Matthew S. Hecht and Jeffrey D. Ullman. "Flow graph Reducibility". In: *Proceedings of the 4th Symposium on Theory of Computing.* STOC'72. ACM, 1972, pp. 238–250.

[Iac+12] M. E. Iacob et al. "From Enterprise Architecture to Business Models and back". In: *Software & Systems Modeling* (2012), pp. 1–25.

[ISO96] ISO. *Information Technology - Syntactic Metalanguage - Extended BNF.* Tech. rep. ISO, Aug. 1996.

[ISO98] ISO/IEC. *ITU-T X.901 ISO/IEC 10746-1: Information Technology - Open Distributed Processing - Reference Model: Overview.* International Standard V1. 1998.

[Jac02] Daniel Jackson. "Alloy: A Lightweight Object Modelling Notation". In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 11.2 (2002), pp. 256–290.

[JB05] Frédéric Jouault and Jean Bezivin. "Using ATL for Checking Models". In: *Proceedings of the International Workshop on Graph and Model Transformation.* GraMoT'05. 2005.

[JBT06] Frédéric Jouault, Jean Bézivin, and Atlas Team. "KM3: A DSL for Metamodel Specification". In: *Proceedings of the 8th International Conference on Formal Methods for Open Object-based Distributed Systems.* FMOODS'06. Springer-Verlag, 2006, pp. 171–185.

[JI09] Henk Jonkers and Maria-Eugenia Iacob. "Performance and Cost Analysis of Service-oriented Enterprise Architectures". In: *Global Implications of Modern Enterprise Information Systems: Technologies and Applications, IGI Global* (2009).

[JNL07] Pontus Johnson, Lars Nordström, and Robert Lagerström. "Formalizing Analysis of Enterprise Architecture". In: *Enterprise Interoperability.* Ed. by Guy Doumeingts et al. Springer-Verlag, 2007, pp. 35–44.

[Joh+07a] Pontus Johnson et al. "Enterprise Architecture Analysis with extended Influence Diagrams". In: *Information Systems Frontiers* 9.2-3 (2007), pp. 163–180.

[Joh+07b] P. Johnson et al. "A Tool for Enterprise Architecture Analysis". In: *11th IEEE International Enterprise Distributed Object Computing Conference (EDOC'07).* 2007.

[Joh+13] Pontus Johnson et al. “P2AMF: Predictive, Probabilistic Architecture Modeling Framework”. In: *Enterprise Interoperability*. Lecture Notes in Business Information Processing 144. Springer-Verlag, 2013, pp. 104–117.

[Joh75] Stephen C Johnson. *Yacc: Yet another Compiler-Compiler*. Vol. 32. Bell Laboratories Murray Hill, NJ, 1975.

[Jou+06] Frédéric Jouault et al. “ATL: a QVT-like transformation language”. In: *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*. ACM. 2006, pp. 719–720.

[JSS00] Daniel Jackson, Ian Schechter, and Ilya Shlyakhter. “Alcoa: The Alloy Constraint Analyzer”. In: *Proceedings of the 22nd International Conference on Software Engineering*. ICSE’00. ACM, 2000, pp. 730–733.

[JW91] Kathleen Jensen and Niklaus Wirth. *Pascal User Manual and Report - ISO Pascal standard*. 4th. Springer-Verlag, 1991.

[Kas90] Uwe Kastens. *Übersetzerbau*. Oldenbourg, 1990. isbn: 978-3486207804.

[KBA02] I. Kurtev, J. Bézivin, and M. Akşit. “Technological Spaces: An Initial Appraisal”. In: *CoopIS, DOA’2002 Federated Conferences, Industrial track*. 2002.

[Ken71] K. Kennedy. “A Global Flow Analysis Algorithm”. In: *International Journal of Computer Mathematics* 3.1 (1971), pp. 5–15.

[Kie+14] Julian Kienberger et al. “Analysis and Validation of AUTOSAR Models”. In: *Proceedings of the 2nd International Conference on Model-Driven Engineering and Software Development*. MODELSWARD’14. SciTePress, 2014.

[Kil73] G.A. Kildall. “A unified Approach to global Program Optimization”. In: *Proceedings of the 1st ACM SIGACT-SIGPLAN symposium on Principles of Programming Languages*. ACM. 1973, pp. 194–206.

[KL88] B. Korel and J.W. Laski. “Dynamic Program Slicing”. In: *Information Processing Letters* 29.3 (1988), pp. 155–163.

[Kle09] A. Kleppe. “The Field of Software Language Engineering”. In: *Software Language Engineering* (2009), pp. 1–7.

[Knu68] Donald E. Knuth. “Semantics of Context-Free Languages”. In: *Theory of Computing Systems* 2.2 (June 1968), pp. 127–145.

[Kos06] Rainer Koschke. “Survey of Research on Software Clones”. In: *Duplication, Redundancy, and Similarity in Software*. Vol. 6301. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), 2006.

[KPP06] Dimitrios S Kolovos, Richard F Paige, and Fiona AC Polack. "The Epsilon Object Language (EOL)". In: *Model Driven Architecture–Foundations and Applications*. Springer-Verlag. 2006, pp. 128–142.

[KPP09] Dimitrios S. Kolovos, Richard F. Paige, and Fiona A. C. Polack. "On the Evolution of OCL for Capturing Structural Constraints in Modelling Languages". In: *Rigorous Methods for Software Construction and Analysis*. Vol. 5115. Springer-Verlag, 2009, pp. 204–218.

[Kra12] Joachim Kraus. "Konzeption und Evaluation eines datenflussbasierten Algorithmus zur Clone Detection". MA thesis. University of Augsburg, 2012.

[Kru95] P. Kruchten. "The 4+1 View Model of Architecture". In: *IEEE Software* 12.6 (1995), pp. 42–50.

[KSK09] Uday Khedker, Amitabha Sanyal, and Bageshri Karkare. *Data Flow Analysis: Theory and Practice*. 1st. Boca Raton, FL, USA: CRC Press, Inc., 2009. isbn: 978-0849328800.

[KU76] J.B. Kam and J.D. Ullman. "Global Data flow Analysis and iterative Algorithms". In: *Journal of the ACM (JACM)* 23.1 (1976), pp. 158–171.

[KU77] J.B. Kam and J.D. Ullman. "Monotone Data flow Analysis Frameworks". In: *Acta Informatica* 7.3 (1977), pp. 305–317.

[Kum+08] A. Kumar et al. "Enterprise Interaction Ontology for Change Impact Analysis of Complex Systems". In: *Proceedings of the 3rd Asia-Pacific Services Computing Conference*. APSCC'08. 2008, pp. 303–309.

[Kun08] A. Kunert. "Semi-automatic Generation of Metamodels and Models from Grammars and Programs". In: *Electronic Notes in Theoretical Computer Science* 211 (2008), pp. 111–119.

[Kur+06] I. Kurtev et al. "Model-based DSL Frameworks". In: *Companion to the 21st ACM SIGPLAN symposium on Object-oriented Programming Systems, Languages, and Applications*. ACM. 2006, pp. 602–616.

[Lan12] Marc Lankhorst. *Enterprise Architecture at Work*. German. Berlin: Springer-Verlag, 2012. isbn: 978-3642296505.

[Lau10] Florian Lautenbacher. "Semantic Business Process Modeling: Principles, Design Support and Realization". PhD thesis. University of Augsburg, 2010. isbn: 978-3832294878.

[Lin11] P. Linz. *An Introduction to Formal Languages and Automata*. Jones & Bartlett Learning, 2011.

[LJJ07] B. Langlois, C.E. Jitia, and E. Jouenne. "DSL Classification". In: *Proceedings of the 7th Workshop on Domain Specific Modeling*. OOPSLA'07. 2007.

[Loh13] Philipp Lohmüller. "Extending GMF to Enable Natural Language Controlled Applications". MA thesis. University of Augsburg, 2013.

[LS07] Ruopeng Lu and Shazia Sadiq. "A Survey of Comparative Business Process Modeling Approaches". In: *Proceedings of the 10th International Conference on Business Information Systems*. BIS'07. Springer-Verlag, 2007, pp. 82–94.

[LSB14a] Melanie Langermeier, Christian Saad, and Bernhard Bauer. "A unified Framework for Enterprise Architecture Analysis". In: *Proceedings of the Enterprise Model Analysis Workshop in the context of the 18th Enterprise Computing Conference*. EDOC'14. 2014.

[LSB14b] Melanie Langermeier, Christian Saad, and Bernhard Bauer. "Context-sensitive Impact Analysis for Enterprise Architecture Management". In: *Proceedings of the 4th International Symposium on Business Modeling and Software Design*. BMSD'14. 2014.

[LT79] T. Lengauer and R.E. Tarjan. "A fast Algorithm for finding Dominators in a Flowgraph". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 1.1 (1979), pp. 121–141.

[Mat+12] Florian Matthes et al. *EAM KPI Catalog v 1.0*. Tech. rep. Technical University Munich, 2012.

[MB06] Slavisa Markovic and Thomas Baar. "An OCL Semantics Specified with QVT". In: *Proceedings of the 9th International Conference on Model Driven Engineering Languages and Systems*. Vol. 4199. MoDELS'06. Springer-Verlag, 2006, pp. 661–675.

[MFJ05] Pierre-Alain Muller, Franck Fleurey, and Jean-Marc Jézéquel. "Weaving Executability into Object-oriented Meta-languages". In: *Proceedings of the 8th International Conference on Model Driven Engineering Languages and Systems*. MoDELS'05. Springer-Verlag, 2005, pp. 264–278.

[MGB04] Tiago Massoni, Rohit Gheyi, and Paulo Borba. "A UML Class Diagram Analyzer". In: *Proceedings of the 3rd International Workshop on Critical Systems Development with UML*. 2004, pp. 143–153.

[MH03] Eva Magnusson and Görel Hedin. "Circular Reference Attributed Grammars - Their Evaluation and Applications". In: *Electronic Notes on Theoretical Computer Science* 82.3 (2003).

[MH05] P.A. Muller and M. Hassenforder. "HUTN as a Bridge between Modelware and Grammarware - An Experience Report". In: *Proceedings of the 4th Workshop in Software Model Engineering in the context of the 8th International Conference on Model Driven Engineering Languages and Systems (MoDELS)*. WiSME'05. 2005.

[MID14] MID GmbH. *MID Innovator for Enterprise Architects*. 2014. url: http://www.mid.de/produkte/innovator-enterprise-modeling.html (visited on 02/28/2014).

[Min12] Pascal Minnerup. "Models in the Development Process for Parallelizing Embedded Systems". MA thesis. University of Augsburg, 2012.

[MLA10] Jeff McAffer, Jean-Michel Lemieux, and Chris Aniszczyk. *Eclipse Rich Client Platform*. 2nd. Addison-Wesley, 2010. isbn: 978-0321603784.

[MLZ06] J. Mendling, K. Bisgaard Lassen, and Uwe Zdun. “Transformation Strategies between Block-Oriented and Graph-Oriented Process Modelling Languages”. In: *Multikonferenz Wirtschaftsinformatik 2006 (MKWI 2006)*. Vol. 3. 2. GITO-Verlag Berlin, 2006, pp. 297–312.

[MM06] H. Malgouyres and G. Motet. “A UML Model Consistency Verification Approach based on Meta-modeling Formalization”. In: *Proceedings of the ACM Symposium on Applied Computing*. ACM. 2006, pp. 1804–1809.

[Mor98] R. Morgan. *Building an optimizing Compiler*. Digital Press, 1998. isbn: 978-1555581794.

[MR08] Michael zur Muehlen and Jan Recker. “How Much Language Is Enough? Theoretical and Practical Use of the Business Process Modeling Notation”. In: *Proceedings of the International Conference on Advanced Information Systems Engineering*. Vol. 5074. CAiSE’08. Springer-Verlag, 2008, pp. 465–479.

[MV99] Luis Mandel and María Victoria. “On the Expressive Power of OCL”. In: *World Congress on Formal Methods*. Vol. 1708/1999. 1999, p. 713.

[MVA10] Jeff McAffer, Paul VanderLei, and Simon Archer. *OSGi and Equinox: Creating Highly Modular Java Systems*. 1st. Addison-Wesley, 2010. isbn: 978-0321585714.

[NBE12] Per Närman, Markus Buschle, and Mathias Ekstedt. “An Enterprise Architecture Framework for multi-attribute Information Systems Analysis”. In: *Software & Systems Modeling* (2012), pp. 1–32.

[Nie06] Klaus D Niemann. *From Enterprise Architecture to IT Governance*. Springer-Verlag, 2006. isbn: 3834801984.

[NNH99] F. Nielson, H.R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999. isbn: 3540654100.

[ODA08] C. Ouyang, M. Dumas, and W.M.P. van der Aalst. “Pattern-based translation of BPMN process models to BPEL web services”. In: *International Journal of Web Services Research (IJWSR)* 5.1 (2008), pp. 42–62.

[Ode+04] Martin Odersky et al. *An Overview of the Scala Programming Language*. Tech. rep. IC/2004/64. EPFL Lausanne, Switzerland, 2004.

[Ode93] Martin Odersky. “Defining Context-dependent Syntax without using Contexts”. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 15.3 (1993), pp. 535–562.

[OWL09] W3C OWL Working Group. *OWL 2 Web Ontology Language: Document Overview*. Available at http://www.w3.org/TR/owl2-overview/. W3C Recommendation, 2009.

[Par07] Terence Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. Pragmatic Bookshelf, May 2007, p. 376. isbn: 0978739256.

[PP08] M. Pfeiffer and J. Pichler. "A Comparison of Tool support for textual Domain-specific Languages". In: *Proceedings of the 8th Workshop on Domain-Specific Modeling*. OOPSLA'08. 2008, pp. 1–7.

[Pro59] R.T. Prosser. "Applications of Boolean Matrices to the Analysis of Flow Diagrams". In: *Eastern Joint IRE-AIEE-ACM Computer Conference*. ACM. 1959, pp. 133–138.

[RCK09] Chanchal K Roy, James R Cordy, and Rainer Koschke. "Comparison and Evaluation of Code Clone Detection Techniques and Tools: A Qualitative Approach". In: *Science of Computer Programming* 74.7 (2009), pp. 470–495.

[RG98] Mark Richters and Martin Gogolla. "On Formalizing the UML Object Constraint Language OCL". In: *Conceptual Modeling*. Vol. 1507. ER'98. Springer-Verlag, 1998, pp. 449–464.

[Ric53] Henry Gordon Rice. "Classes of Recursively Enumerable Sets and their Decision Problems". In: *Transactions of the American Mathematical Society* 74.2 (1953), pp. 358–366.

[Rod86] F. Rodney. "Automatic Generation of fixed-point-finding Evaluators for circular, but well-defined, Attribute Grammars". In: *Proceedings of the SIGPLAN symposium on Compiler Construction*. SIGPLAN'86. ACM, 1986, pp. 85–98.

[Ros90] Mads Rosendahl. "Abstract Interpretation using Attribute Grammars". In: *Attribute Grammars and their Applications*. Springer-Verlag, 1990, pp. 143–156.

[RW82] Sandra Rapps and Elaine J Weyuker. "Data flow Analysis Techniques for Test Data Selection". In: *Proceedings of the 6th International Conference on Software Engineering*. ICSE'82. IEEE. 1982, pp. 272–278.

[Ryd83] B.G. Ryder. "Incremental Data flow Analysis". In: *Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of Programming Languages*. ACM. 1983, pp. 167–176.

[SAB10] S. Shah, K. Anastasakis, and B. Bordbar. "From UML to Alloy and back again". In: *Models in Software Engineering* (2010), pp. 158–171.

[Sag+89] Shmuel Sagiv et al. "Resolving Circularity in Attribute Grammars with Applications to Data Flow Analysis". In: *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM. 1989, pp. 36–48.

[Sal87] Arto Salomaa. *Formal Languages*. San Diego, CA, USA: Academic Press Professional, Inc., 1987. isbn: 0126157502.

[SB05] A. Staikopoulos and B. Bordbar. "A Metamodel Refinement Approach for Bridging Technical Spaces, a Case Study". In: *Proceedings of the 4th Workshop in Software Model Engineering in the context of the 8th International Conference on Model Driven Engineering Languages and Systems (MoDELS)*. WiSME'05. 2005.

[SB10a] Christian Saad and Bernhard Bauer. "Analyzing Dynamic Models using a Data-flow based Approach". In: *Proceedings of the 1st Doctoral Symposium in the context of the 3rd International Conference on Software Language Engineering*. SLE'10. 2010, p. 37.

[SB10b] Christian Saad and Bernhard Bauer. "Applying Data-flow Analysis to Models - A Novel Approach for Model Analysis". In: *Proceedings of the Spring Simulation Multiconference*. SpringSim'10. ACM, 2010, p. 241.

[SB10c] Christian Saad and Bernhard Bauer. "Data-flow Based Model Analysis". In: *Proceedings of the 2nd NASA Formal Methods Symposium*. Vol. NASA/CP-2010-216215. NFM'10. NASA. 2010, pp. 227–231.

[SB11] Christian Saad and Bernhard Bauer. "The Model Analysis Framework - An IDE for Static Model Analysis". In: *Proceedings of the Industry Track of Software Language Engineering in the context of the 4th International Conference on Software Language Engineering (SLE)*. ITSLE'11. 2011.

[SB13] Christian Saad and Bernhard Bauer. "Data-flow based Model Analysis and its Applications". In: *Proceedings of the 16th International Conference on Model Driven Engineering Languages and Systems*. MoDELS'13. Springer-Verlag, 2013, pp. 707–723.

[SC13] Matthew Stephan and James R Cordy. "A Survey of Model Comparison Approaches and Applications". In: *Proceedings of the 1st International Conference on Model-Driven Engineering and Software Development*. MODELSWARD'13. SciTePress, 2013, pp. 265–277.

[SCH02] W. Shen, K. Compton, and J. Huggins. "A Toolset for supporting UML static and dynamic Model Checking". In: *Computer Software and Applications Conference, 2002. COMPSAC 2002. Proceedings. 26th Annual International*. IEEE. 2002, pp. 147–152.

[SK95] Kenneth Slonneger and Barry Kurtz. *Formal Syntax and Semantics of Programming Languages: A Laboratory Based Approach*. 1st. Boston, MA, USA: Addison-Wesley, 1995.

[SKR13] Sagar Sunkle, Vinay Kulkarni, and Suman Roychoudhury. "Analyzing Enterprise Models using Enterprise Architecture-based Ontology". In: *Model-Driven Engineering Languages and Systems*. Springer-Verlag, 2013, pp. 622–638.

[SLB09] Christian Saad, Florian Lautenbacher, and Bernhard Bauer. “An Attribute-based Approach to the Analysis of Model Characteristics”. In: *Proceedings of the 1st International Workshop on Future Trends of Model-Driven Development in the context of the 11th International Conference on Enterprise Information Systems (ICEIS)*. Vol. 9. FT-MDD’09. 2009.

[Slo08] Tony Sloane. “Experiences with Domain-specific Language Embedding in Scala”. In: *Domain-Specific Program Development* (2008).

[Soe+10] Mathias Soeken et al. “Verifying UML/OCL models using Boolean Satisfiability”. In: *Proceedings of the Conference on Design, Automation and Test in Europe*. European Design and Automation Association. 2010, pp. 1341–1344.

[Ste+09] Dave Steinberg et al. *EMF: Eclipse Modeling Framework*. 2. Boston, MA: Addison-Wesley, 2009. isbn: 978-0321331885.

[SVC06] Thomas Stahl, Markus Voelter, and Krzysztof Czarnecki. *Model-Driven Software Development: Technology, Engineering, Management*. John Wiley & Sons, 2006. isbn: 0470025700.

[SZ08] L. Shan and H. Zhu. “A formal descriptive Semantics of UML”. In: *Formal Methods and Software Engineering* (2008), pp. 375–396.

[Tar72] Robert Tarjan. “Depth-first Search and Linear Graph Algorithms”. In: *SIAM Journal on Computing* 1.2 (1972), pp. 146–160.

[The11] The Open Group. *TOGAF Version 9.1*. Van Haren Publishing, 2011.

[Thi09] Krishnaprasad Thirunarayan. “Attribute Grammars and their Applications”. In: *Encyclopedia of Information Science and Technology* (2009), pp. 268–273.

[Uba+11] Reina Uba et al. “Clone Detection in Repositories of Business Process Models”. In: *Proceedings of the 9th International Conference on Business Process Management*. BPM’09. Springer-Verlag, 2011, pp. 248–264.

[UL07] Mark Utting and Bruno Legeard. *Practical Model-based Testing: A Tools Approach*. Morgan Kaufmann Publishers Inc., 2007. isbn: 978-0080466484.

[US 10] U.S. Department of Defense. *The DoDAF Architecture Framework Version 2.02*. 2010. url: `http://dodcio.defense.gov/dodaf20.aspx` (visited on 04/09/2013).

[Van+07] Irene Vanderfeesten et al. “Quality Metrics for Business Process Models”. In: *BPM and Workflow Handbook*. Future Strategies Inc., 2007, pp. 179–190.

[Var02] Dániel Varró. “A formal Semantics of UML Statecharts by Model Transition Systems”. In: *Proceedings of the 1st International Conference on Graph Transformation*. ICGT’02. Springer-Verlag, 2002, pp. 378–392.

[VHW03] Wil M. P. Van Der Aalst, Arthur H. M. Ter Hofstede, and Mathias Weske. “Business Process Management: A Survey”. In: *Proceedings of the 1st International Conference on Business Process Management*. BPM’03. Springer-Verlag, 2003, pp. 1–12.

[VJ00] Mandana Vaziri and Daniel Jackson. “Some Shortcomings of OCL, the Object Constraint Language of UML”. In: *Proceedings of the 34th International Conference on Technology of Object-Oriented Languages and Systems*. TOOLS’00. IEEE, 2000, pp. 555–562.

[VTM08] Kostas Vergidis, Ashutosh Tiwari, and Basim Majeed. “Business Process Analysis and Optimization: Beyond Reengineering”. In: *IEEE Transactions on Systems, Man, and Cybernetics, Part C* 38.1 (2008), pp. 69–82.

[VVL07] Jussi Vanhatalo, Hagen Völzer, and Frank Leymann. “Faster and more focused Control-Flow Analysis for Business Process Models through SESE Decomposition”. In: *Proceedings of the 5th international conference on Service-Oriented Computing*. ICSOC ’07. Springer-Verlag, 2007, pp. 43–55.

[VW63] V. Vyssotsky and P Wegner. *A Graph theoretical Fortran Source Language Analyzer*. Technical Report (unpublished). Bell Laboratories, Murray Hill NJ, 1963.

[WFM96] WFMC. *Workflow Management Coalition Terminology and Glossary (WFMC-TC-1011)*. Technical Report. Workflow Management Coalition, Brussels, 1996.

[WG98] William McCastline Waite and Gerhard Goos. *Compiler Construction*. Springer-Verlag, 1998. isbn: 978-3540642565.

[Wik08] Wikipedia. *Meta Object Facility*. 2008. url: `http://de.wikipedia.org/wiki/Meta_Object_Facility` (visited on 07/04/2008).

[Wil10] Edward D. Willink. “Re-engineering Eclipse MDT/OCL for Xtext”. In: *Electronic Communications of the ECEASST* 36 (2010).

[WIM08] Tabinda Waheed, Muhammad Zohaib Iqbal, and Zafar I. Malik. “Data Flow Analysis of UML Action Semantics for Executable Models”. In: *Proceedings of the 4th European Conference on Model Driven Architecture: Foundations and Applications*. Vol. 5095. ECMDA-FA’08. Springer-Verlag, 2008, pp. 79–93.

[Wir63] N. Wirth. “Programming in Oberon”. In: *Communications of the ACM* 6 (1963), pp. 1–17.

[Wir77] Niklaus WIRTH. “What can we do about the unnecessary diversity of notation for syntactic definitions?” In: *Communications of the ACM* 20 (11 Nov. 1977), pp. 822–823.

[WK03] Jos WARMER and Anneke KLEPPE. *The Object Constraint Language: Getting Your Models Ready for MDA*. 2nd ed. Boston, MA, USA: Addison-Wesley, 2003. ISBN: 0321179366.

[WK05] Manuel WIMMER and Gerhard KRAMLER. “Bridging Grammarware and Modelware”. In: *Proceedings of the Satellite Events of the 8th International Conference on Model Driven Engineering Languages and Systems*. Vol. 3844. MoDELS’05. Springer-Verlag, 2005, pp. 159–168.

[WM95] Reinhard WILHELM and Dieter MAURER. *Compiler Design*. International Computer Science Series. Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN: 0201422905.

[Woh+06] Petia WOHED et al. “On the Suitability of BPMN for Business Process Modelling”. In: *Proceedings of the 4th International Conference on Business Process Management*. Vol. 4102. BPM’06. Springer-Verlag, 2006, pp. 161–176.

[Zsc+10] Steffen ZSCHALER et al. “Domain-Specific Metamodelling Languages for Software Language Engineering”. In: *Proceedings of the 2nd International Conference on Software Language Engineering*. SLE’09. Springer-Verlag, 2010, pp. 334–353.

Acronyms

AG	Attribute Grammar
API	Application Programming Interface
AST	Abstract Syntax Tree
AUTOSAR	AUTomotive Open System ARchitecture
BFS	Breadth-first Search
BPM	Business Process Modeling
BPMN	Business Process Modeling Notation
CC	Compiler Construction
CFG	Context-free Grammar
CFL	Context-free Language
CRAG	Circular Reference Attributed Grammar
CST	Concrete Syntax Tree
DFA	Data-flow Analysis
DFS	Depth-first Search
DFST	Depth-first Spanning Tree
DSL	Domain-specific Language
DSM	Domain-specific Model
EAM	Enterprise Architecture Management
EBNF	Extended Backus-Naur Form
EMF	Eclipse Modeling Framework
EMOF	Essential Meta-Object Facility
EOL	Epsilon Object Language
EPL	Eclipse Public Licence
EVL	Epsilon Validation Language
GEF	Graphical Editing Framework
GMF	Graphical Modeling Framework
GPL	GNU General Public License
IDE	Integrated Development Environment
JDT	Java Development Tools
JWT	Java Workflow Tooling
KM3	Kernel Meta Meta Model
KPI	Key Performance Indicator
M2M	Model to Model
M2T	Model to Text
MAF	Model Analysis Framework
MBT	Model-based Testing
MDA	Model-driven Architecture
MDE	Model-driven Engineering

MDSD	Model-driven Software Development
MOF	Meta Object Facility
OCL	Object Constraint Language
OMG	Object Management Group
PIM	Platform Independent Model
PSM	Platform Specific Model
QVT	Query/View/Transformation
RAG	Reference Attributed Grammar
RCP	Rich Client Platform
RUP	Rational Unified Process
SCC	Strongly Connected Component
SE-DSNL	Semantically Enhanced Domain-Specific Natural Language
SESE	Single Entry Single Exit
SLE	Software Language Engineering
SSA	Static Single-Assignment
TAC	Three-Address Code
TMF	Textual Modeling Framework
TS	Technological Space
UML	Unified Modeling Language
XMI	XML Metadata Interchange
XML	Extensible Markup Language

List of Figures

List of Tables

List of Algorithms

Appendix A. Basics

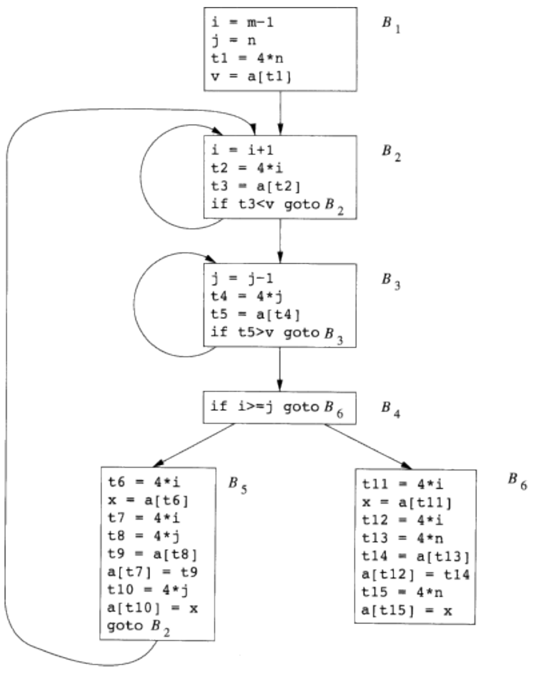
A.1. Intermediate and Control-flow Representation of Programs

```
void quicksort(int m, int n)
    /* recursively sorts a[m] through a[n] */
{
    int i, j;
    int v, x;
    if (n <= m) return;
    /* fragment begins here */
    i = m-1; j = n; v = a[n];
    while (1) {
        do i = i+1; while (a[i] < v);
        do j = j-1; while (a[j] > v);
        if (i >= j) break;
        x = a[i]; a[i] = a[j]; a[j] = x; /* swap a[i], a[j] */
    }
    x = a[i]; a[i] = a[n]; a[n] = x; /* swap a[i], a[n] */
    /* fragment ends here */
    quicksort(m,j); quicksort(i+1,n);
}
```

(a) Quick sort algorithm

```
 (1)   i = m-1              (16)   t7 = 4*i
 (2)   j = n                (17)   t8 = 4*j
 (3)   t1 = 4*n             (18)   t9 = a[t8]
 (4)   v = a[t1]            (19)   a[t7] = t9
 (5)   i = i+1              (20)   t10 = 4*j
 (6)   t2 = 4*i             (21)   a[t10] = x
 (7)   t3 = a[t2]           (22)   goto (5)
 (8)   if t3<v goto (5)     (23)   t11 = 4*i
 (9)   j = j-1              (24)   x  = a[t11]
(10)   t4 = 4*j             (25)   t12 = 4*i
(11)   t5 = a[t4]           (26)   t13 = 4*n
(12)   if t5>v goto (9)     (27)   t14 = a[t13]
(13)   if i>=j goto (23)    (28)   a[t12] = t14
(14)   t6 = 4*i             (29)   t15 = 4*n
(15)   x = a[t6]            (30)   a[t15] = x
```

(b) Three-Address Code representation



(c) Control-flow graph with basic blocks

Figure A.1.: Different representations of the quick sort algorithm [Aho+06].

A.2. Data-flow Frameworks

	Available expressions	*Dominators*
Domain	Sets of expressions	The power set of N
Direction	Forwards	Forwards
Transfer Function	$e\_gen_B \cup e\_kill_B$	$f_B(x) = x \cup \{B\}$
Boundary	OUT[`ENTRY`]=$\emptyset$	OUT[`ENTRY`]={`ENTRY`}
Meet ($\wedge$)	$\cap$	$\cap$
Equations	$OUT[B] = f_B(IN[B])$ $IN[B] = \bigwedge_{P,pred(B)} OUT[P]$	$OUT[B] = f_B(IN[B])$ $IN[B] = \bigwedge_{P,pred(B)} OUT[P]$
Initialize	OUT[B]=U	OUT[B]=N

Table A.1.: Global data-flow analysis frameworks [Aho+06].

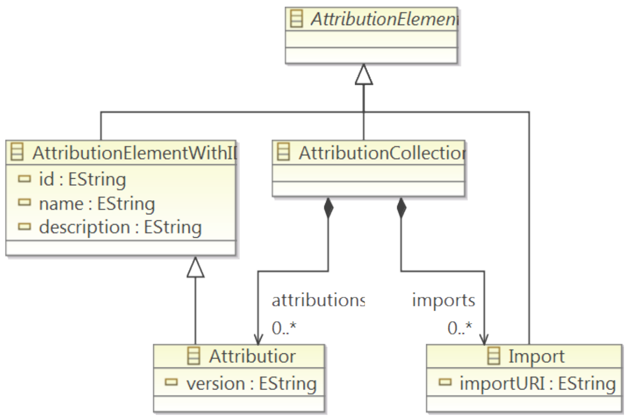
Appendix B.

Integrating Flow Analysis with the Modeling Domain

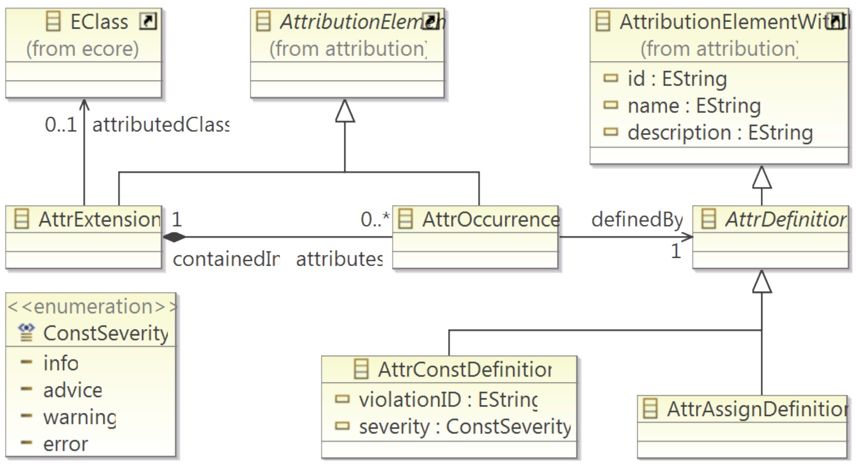
B.1. Attribution Metamodel

Detailed view of the components of the packages of the attribution metamodel AttrMM.

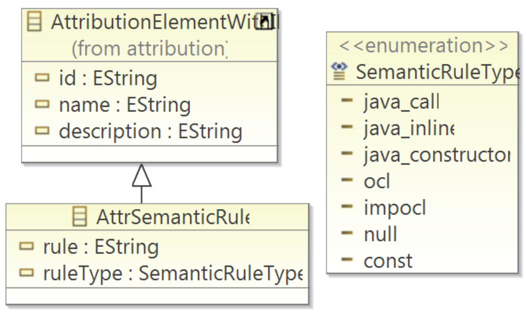
The Attribution package



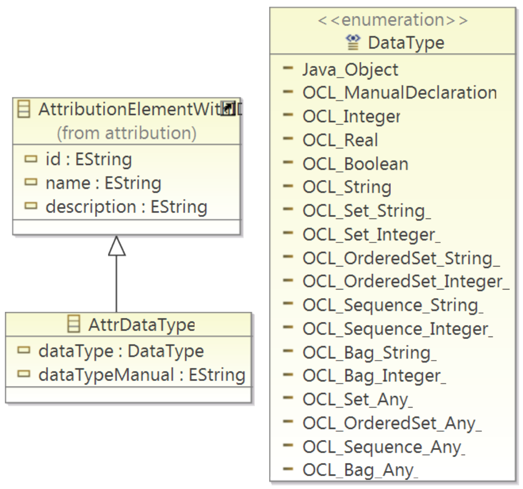
The Attributes package



The Semanticrules package



The Datatypes package



B.2. Alternative Worklist Algorithm for the DFA Solver Framework

Algorithm 44 Alternative worklist algorithm for the DFA solver framework

```
1: Map⟨AttrInstance, Set⟨AttrInstance⟩⟩ outputDependencies = {}      ▷ dependencies
2: OrderedSet⟨AttrInstance⟩ worklist = {}                            ▷ the main worklist
3: Set⟨AttrInstance⟩ processedInstances = {}    ▷ instances that have been invoked once
4: Stack⟨AttrInstance⟩ callStack = {}          ▷ stack that records recursive invocations
```

```
1: function AnalysisEntryPoint(Set⟨AttrInstance⟩ selectedInstances)
2:     worklist.addAll(selectedInstances)          ▷ set up worklist with requested instances
3:     repeat                                                  ▷ process worklist entries
4:         AttrInstance instance = worklist.remove(0)    ▷ pick and remove worklist entry
5:         Object oldValue = instance.value                          ▷ remember old value
6:         processedInstances.add(instance)                     ▷ mark instance as processed
7:         callStack.push(instance)                             ▷ push instance on callstack
8:         invoke(instance)                                         ▷ invoke iteration rule
9:         callStack.pop()                                 ▷ remove instance from callstack
10:        if (not oldValue == instance.value) then                    ▷ if value changed...
11:            worklist.addAll(outputDependencies[instance])          ▷ ...update worklist
12:    until (worklist.size == 0)                            ▷ repeat until worklist is empty
```

```
1: function CallbackHandler(AttrInstance requestingInstance, requestedInstance,
                            boolean dynamicallyDiscovered)
2:     ▷ add requesting instance to output dependency set of requested instance
3:     outputDependencies[requestedInstance] += requestingInstance
4:     if (callingStack.contains(requestedInstance)) then           ▷ detect cyclic dependency
5:         if (not requestedInstance.isInitialized()) then      ▷ initialize cyclic dependency
6:             init(requestedInstance)
7:         worklist.addAll(outputDependencies[requestedInstance])                ▷ reevaluate
8:     else if (!processedInstances.contains(requestedInstance)) then
9:         processedInstances.add(requestedInstance)            ▷ mark instance as processed
10:        callStack.push(requestedInstance)                    ▷ push instance on callstack
11:        invoke(requestedInstance)                                          ▷ invoke rule
12:        callStack.pop()                                 ▷ remove instance from callstack
```

This is an alternative version of a worklist algorithm (Algorithm 12) for the solver framework presented in Section 6.5. This algorithm relies on a callstack to record recursively triggered invocations. If an instance is already on the stack [4], a cyclic dependency has been discovered. In this case, the respective instance is initialized [5-6] and its output dependencies are added to the worklist [7].

Appendix C.

Case Studies and Applications

C.1. Business Process Analysis

This section contains additional material for the case study presented in Section 10.1.

C.1.1. Rule Implementations using Imperative OCL

Control-flow Analysis

Rule implementations in Imperative OCL for the attribution cfg_flowset:

```
query ActivityNode::ActivityNode__allPredecessorsMax() : Set(OclAny)
{
    var directPred:Set(OclAny):=self._in.source->asSet();
    var transitivePred:Set(OclAny):=
        self._in.source.allPredecessorsMax()->asSet();

    return directPred->union(transitivePred)
}

query ActivityNode::ActivityNode__allPredecessorsMin() : Set(OclAny)
{
    if (self._in->size()=0) then return Set{} endif;

    var res : Set(OclAny):= null;
    self._in.source->forEach(predNode)
    {
        -- access predMin at predecessor Node
        var predValue:Set(OclAny) :=
            predNode.allPredecessorsMin()->including(predNode);

        if (predValue->includes("ref")) then predValue :=
            self.container().oclAsType(Activity).nodes endif;

        -- init on first time
        if (res = null) then res := predValue endif;

        -- create intersection
        res := res->intersection(predValue);
    };

    return res
}
```

Rule implementations in Imperative OCL for the attribution cfg_scc:

```
query ActivityNode::ActivityNode__sccID() : Integer
{
    var self_pred : Set(OclAny) := self.allPredecessorsMax();

    if (self_pred->includes("ref")) then return -1 endif;
    if (self_pred->includes(self)) then return self_pred->hashCode() endif;

    return 0
}


query ActivityNode::ActivityNode__sccObjects() : Set(OclAny)
{
    var sccID : Integer := self.sccID();

    if (sccID=-1) then return Set{"ref"} endif;
    if (sccID=0) then return Set{} endif;

    var nodes : Set(OclAny) := Set{self};
    self._in.source->forEach(predNode)
    {
        if (sccID=predNode.sccID()) then
            nodes := nodes->union(predNode.sccObjects())
        endif
    };

    return nodes->excluding("ref")
}
```

Rule implementations in Imperative OCL for the attribution cfg_ports:

```
query ActivityNode::ActivityNode__sccInEdges() : Set(OclAny)
{
    var sccID : Integer := self.sccID();
    if (sccID=0) then return Set{} endif;

    var edges : Set(OclAny) := Set{};
    self._in->forEach(predEdge)
    {
        if (sccID = predEdge.source.sccID()) then
            edges := edges->union(predEdge.source.sccInEdges())
        else
            edges := edges->including(predEdge)
        endif
    };

    return edges->excluding("ref")
}
```

```
query ActivityNode::ActivityNode__sccInPorts() : Set(OclAny)
{
    var sccID : Integer := self.sccID();
    if (sccID=0) then return Set{} endif;

    var edges : Set(OclAny) := Set{};
    self._in->forEach(predEdge)
    {
        if (sccID = predEdge.source.sccID()) then
            edges := edges->union(predEdge.source.sccInPorts())
        else
            edges := edges->including(self)
        endif
    };

    return edges->excluding("ref")
}
```

Definition/Usage of Data Objects

Rule implementation in Imperative OCL for the computation of minimal data availability at ActivityNodes:

```
query ActivityNode::ActivityNode__dataMinAvailable() : Set(OclAny)
{
    -- if in>0 check if there are only domain values
    var domain : Boolean := self._in.source->size() > 0;

    var res : Set(OclAny):= Set{};
    self._in.source->forEach(predNode)
    {
        -- access predMin at predecessor Node
        var predValue:Set(OclAny) := predNode.dataMinAvailable();

        if (not predValue->includes("ref")) then
        {
            -- true value -> throw away value domain
            domain := false;

            -- create intersection
            if (self.oclIsKindOf(JoinNode)) then {
                res := res->union(predValue)
            }
            else {
                if (res->size() = 0) then res := predValue endif;
                res := res->intersection(predValue)
            }
            endif;
        }
        endif;
    };

    -- only domain values -> return value domain
    if (domain) then return Set{"ref"} endif;

    -- add locally created outputs
    return res->union(self.dataOutputs())
}
```

Token Analysis

Rule implementation in Imperative OCL for the computation of the initial tokenset at ActivityEdges:

```
query ActivityEdge::ActivityEdge__tokensInitial() : Set(OclAny)
{
    if (self.source = null) then return Set{}  endif;

    -- the scc_id of the source node
    var source_sccID : Integer = self.source.sccID();

    -- the scc_id of the edge
    var sccID : Integer := 0;
    if (source_sccID = self.target.sccID()) then sccID := source_sccID endif;

    -- outgoing edges at source node
    var sourceOutgoing : Integer = self.source._out->size();

    -- incoming token set
    var tokens : Set(Any) := Set{};
    self.source._in->forEach(incomingEdge)
    {
        tokens := tokens->union(incomingEdge.tokensInitial())
    };

    if (sccID=0 and sourceOutgoing>1)  then
    {
        var newToken:tokenflow::Token := new tokenflow::Token();
        newToken.targetNode := self.target.oclAsType(ecore::EObject);
        if (source_sccID=0) then
        {
            newToken.splitNode := self.source;
            newToken.outgoingCount := sourceOutgoing;
        }
        else
        {
            newToken.splitNode := source_sccID;
            newToken.outgoingCount := self.source.sccOutEdges()->size();
        }
        endif;
        tokens := tokens->including(newToken);
    }
    endif;

    return tokens;
}
```

Rule implementation in Imperative OCL for the computation of the final tokenset at ActivityEdges:

```
query ActivityEdge::ActivityEdge__tokensFinal() : Set(OclAny)
{
    var tokensInitial : Set(OclAny) := self.tokensInitial();
    var tokens : Set(OclAny) := Set{}->union(tokensInitial);
    var originMap : Dict(OclAny, Set(OclAny)) := Dict{};

    tokensInitial->forEach(t)
    {
        -- the current token
        var token : tokenflow::Token := t.oclAsType(tokenflow::Token);

        -- the tokenset with the same splitnode
        var entry : Set(OclAny) := originMap->get(token.splitNode);
        if (entry = null) then entry := Set{} endif;

        -- add current token to the tokenset
        entry := entry->including(token);
        originMap->put(token.splitNode, entry);

        -- if enough tokens in set -> remove
        if (entry->size() = token.outgoingCount) then
            tokens := tokens-entry
        endif;
    };

    return tokens;
}
```

C.1.2. IBM Fast Heuristics on Tokenflow Components

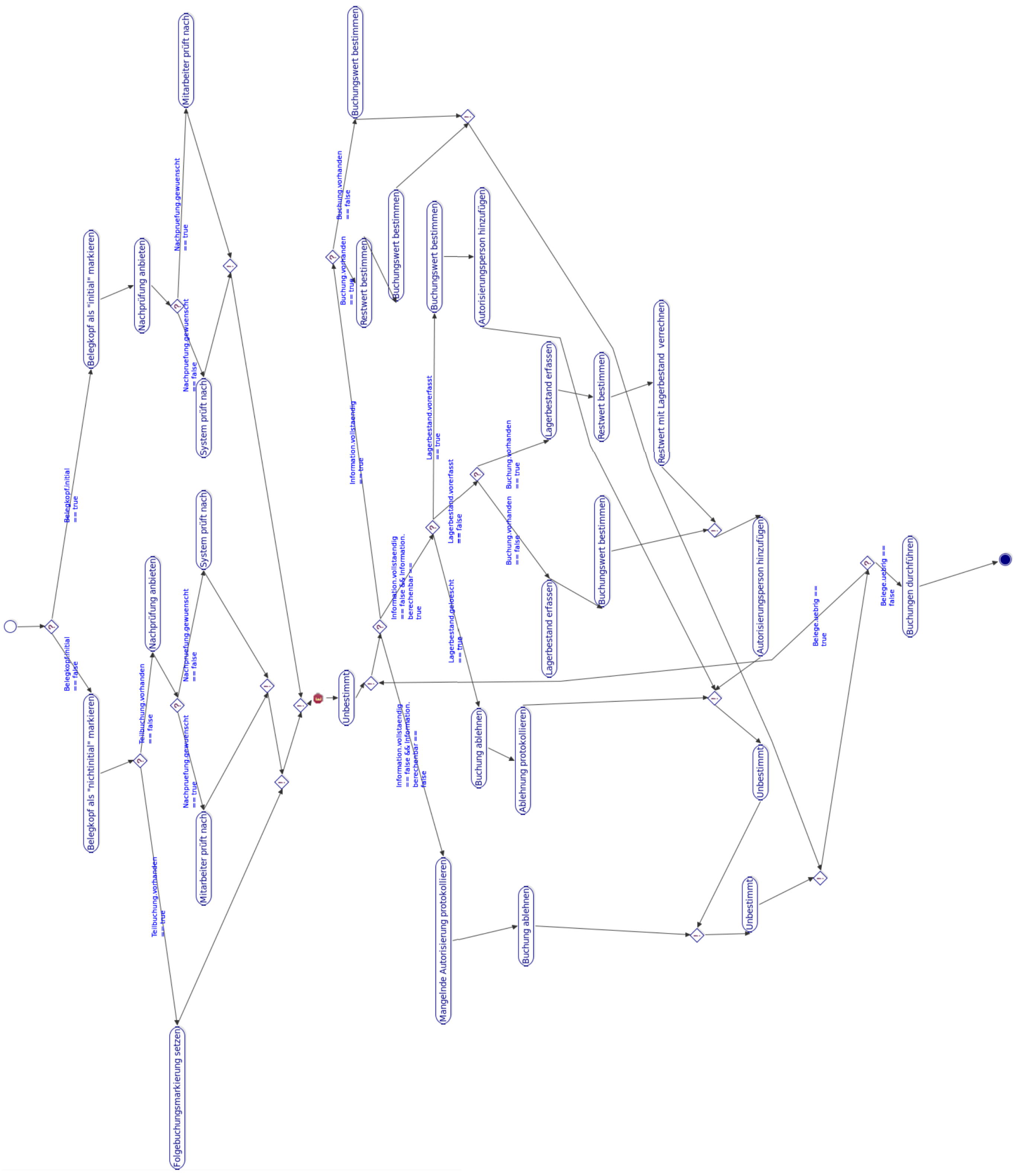
Implementation of the IBM Fast Heuristics approach. This code runs as a MAF macro after component detection has been performed based on the DFA tokenflow algorithm to classify the components according to their structural properties.

```
public void validateComponent(Component component)
{
    // RECURSIVE: validate subcomponents and check their type
    boolean unsoundChild = false;
    boolean unknownChild = false;
    for (Component subComponent : component.getSubComponents())
    {
        // recursively validate subcomponent
        validateComponent(subComponent);

        // store child type info
        if (subComponent.getValidResult().equals(validationResult.UNSOUND))
            unsoundChild = true;
        else if (subComponent.getValidResult().equals(validationResult.UNKNOWN))
            unknownChild = true;
    }

    // classify component type
    classifyComponent(component);

    // determine result status of component
    // a) child overwrites local status
    if (unsoundChild)
        component.setValidResult(validationResult.UNSOUND);
    else if (unknownChild)
        component.setValidResult(validationResult.UNKNOWN);
    else
    {
        // b) set local status according to classification result
        if (component.getValidClassification().equals(
                validationClassification.COMPLEXUNSOUNDDEADLOCK)
                || component.getValidClassification().equals(
                    validationClassification.COMPLEXUNSOUNDLACKOFSYNC))
            component.setValidResult(validationResult.UNSOUND);
        else if (component.getValidClassification().equals(
                validationClassification.COMPLEXUNKNOWN))
            component.setValidResult(validationResult.UNKNOWN);
        else
            component.setValidResult(validationResult.SOUND);
    }
}
public static void classifyComponent(Component component)
{
    // get all child nodes
    Set<EObject> childNodes = new HashSet<EObject>(component.getTrivialNodes());
    childNodes.addAll(component.getSccInputPorts());
    childNodes.addAll(component.getSccOutputPorts());

    // collect information from child nodes
    Set<EObject> decisionNodes = new HashSet<EObject>();
    Set<EObject> mergeNodes = new HashSet<EObject>();
    Set<EObject> forkNodes = new HashSet<EObject>();
    Set<EObject> joinNodes = new HashSet<EObject>();
    for (EObject childNode : childNodes)
        if (childNode.eClass().getName().equals(JWTConstants.EC_DECISIONNODE))
            decisionNodes.add(childNode);
        else if (childNode.eClass().getName().equals(JWTConstants.EC_MERGENODE))
            mergeNodes.add(childNode);
        else if (childNode.eClass().getName().equals(JWTConstants.EC_JOINNODE))
            joinNodes.add(childNode);
        else if (childNode.eClass().getName().equals(JWTConstants.EC_FORKNODE))
            forkNodes.add(childNode);
    // contains cycle
    boolean containsSCC = false;
    for (Component subComponent : component.getSubComponents())
        if (componentType.SCC.equals(subComponent.getType()))
        {
            containsSCC = true;
            break;
        }

    // input/output node information
    EObject inEdge = component.getInputEdges().get(0);
    EObject outEdge = component.getOutputEdges().get(0);
    boolean entryIsIncomingOfDecision = decisionNodes.size() == 1
            && EMFDynamicUtils.getEReferenceValues(decisionNodes.iterator().next(), "in")
                    .contains(inEdge);
    boolean exitIsOutgoingOfMerge = mergeNodes.size() == 1
            && EMFDynamicUtils.getEReferenceValues(mergeNodes.iterator().next(), "out")
                    .contains(outEdge);
    boolean entryIsIncomingOfMerge = mergeNodes.size() == 1
            && EMFDynamicUtils.getEReferenceValues(mergeNodes.iterator().next(), "in")
                    .contains(inEdge);
    boolean exitIsOutgoingOfDecision = decisionNodes.size() == 1
            && EMFDynamicUtils.getEReferenceValues(decisionNodes.iterator().next(), "out")
                    .contains(outEdge);
    boolean entryIsIncomingOfFork = forkNodes.size() == 1
            && EMFDynamicUtils.getEReferenceValues(forkNodes.iterator().next(), "in").contains(
                    inEdge);
    boolean exitIsOutgoingOfJoin = joinNodes.size() == 1
            && EMFDynamicUtils.getEReferenceValues(joinNodes.iterator().next(), "out")
                    .contains(outEdge);
    // apply heuristics

    // well-structured
    if (decisionNodes.size() == 0 && mergeNodes.size() == 0 && forkNodes.size() == 0
            && joinNodes.size() == 0)
        component.setValidClassification(validationClassification.WELLSTRUCTUREDSEQUENCE);
    else if (decisionNodes.size() == 1 && mergeNodes.size() == 1 && forkNodes.size() == 0
            && joinNodes.size() == 0 && entryIsIncomingOfDecision && exitIsOutgoingOfMerge)
        component
                .setValidClassification(validationClassification.WELLSTRUCTUREDSEQUENTIALBRANCHING);
    else if (decisionNodes.size() == 1 && mergeNodes.size() == 1 && forkNodes.size() == 0
            && joinNodes.size() == 0 && entryIsIncomingOfMerge && exitIsOutgoingOfDecision)
        component.setValidClassification(validationClassification.WELLSTRUCTUREDCYCLE);
    else if (decisionNodes.size() == 0 && mergeNodes.size() == 0 && forkNodes.size() == 1
            && joinNodes.size() == 1 && entryIsIncomingOfFork && exitIsOutgoingOfJoin)
        component
                .setValidClassification(validationClassification.WELLSTRUCTUREDCONCURRENTBRANCHING);
    else
    {
        // not well-structured
        if (decisionNodes.size() == 0 && mergeNodes.size() == 0 && !containsSCC)
            component.setValidClassification(validationClassification.UNSTRUCTUREDCONCURRENT);
        else if (forkNodes.size() == 0 && joinNodes.size() == 0)
            component.setValidClassification(validationClassification.UNSTRUCTUREDSEQUENTIAL);
        else
        {
            // complex
            // at least one decision, but no merges
            boolean case1a = !containsSCC && decisionNodes.size() > 0 && mergeNodes.size() == 0;
            // at least one merge, but no decisions
            boolean case1b = !containsSCC && mergeNodes.size() > 0 && decisionNodes.size() == 0;
            // at least one fork, but no joins
            boolean case2a = !containsSCC && forkNodes.size() > 0 && joinNodes.size() == 0;
            // at least one join, but no forks
            boolean case2b = !containsSCC && joinNodes.size() > 0 && forkNodes.size() == 0;
            boolean case3a = containsSCC && decisionNodes.size() == 0 && mergeNodes.size() == 0;
            boolean case3b = containsSCC && decisionNodes.size() > 0 && mergeNodes.size() == 0;
            boolean case3c = containsSCC && decisionNodes.size() == 0 && mergeNodes.size() > 0;

            if (case1a || case2b || case3a || case3b)
                component
                        .setValidClassification(validationClassification.COMPLEXUNSOUNDDEADLOCK);
            else if (case1b || case2a || case3c)
                component
                        .setValidClassification(validationClassification.COMPLEXUNSOUNDLACKOFSYNC);
            else
                component.setValidClassification(validationClassification.COMPLEXUNKNOWN);
        }
    }
}
```

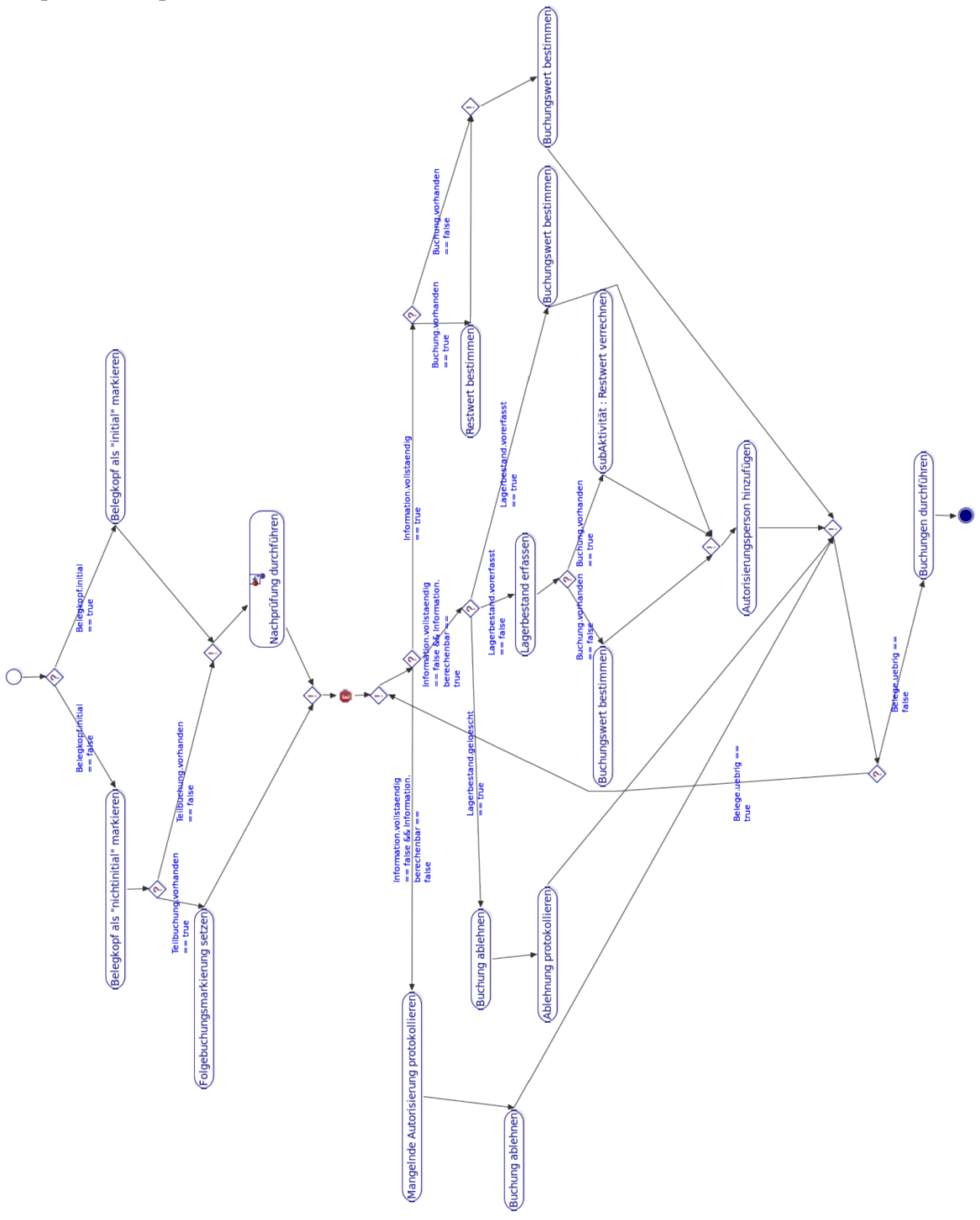
C.1.3. Model Clone Detection

Model clone detection applied to improve a real world business process [Kra12].

Original process


Belegkopf als "initial" markieren
Belegkopf als "nichtinitial" markieren
Nachprüfung anbieten
Mitarbeiter prüft nach
System prüft nach
Folgebuchungsmarkierung setzen
Buchungswert bestimmen
Restwert bestimmen
Autorisierungsperson hinzufügen
Lagerbestand erfassen
Restwert mit Lagerbestand verrechnen
Unbestimmt
Buchung ablehnen
Ablehnung protokollieren
Mangelnde Autorisierung protokollieren
Buchungen durchführen
Belegkopf.initial == true
Belegkopf.initial == false
Nachpruefung.gewuenscht == true
Nachpruefung.gewuenscht == false
Teilbuchung.vorhanden == false
Teilbuchung.vorhanden == true
Buchung.vorhanden == false
Buchung.vorhanden == true
Information.vollstaendig == true
Information.vollstaendig == false && Information.berechenbar == true
Information.vollstaendig == false && Information.berechenbar == false
Lagerbestand.vorerfasst == true
Lagerbestand.vorerfasst == false
Lagerbestand.geloescht == true
Belege.uebrig == false
Belege.uebrig == true

Optimized process


Belegkopf.initial == true
Belegkopf.initial == false
Belegkopf als "initial" markieren
Belegkopf als "nichtinitial" markieren
Teilbuchung.vorhanden == false
Teilbuchung.vorhanden == true
Folgebuchungsmarkierung setzen
Nachprüfung durchführen
Information.vollstaendig == true
Information.vollstaendig == false && Information.berechenbar == true
Information.vollstaendig == false && Information.berechenbar == false
Mangelnde Autorisierung protokollieren
Buchung ablehnen
Buchung.vorhanden == false
Buchung.vorhanden == true
Restwert bestimmen
Buchungswert bestimmen
Buchungswert bestimmen
Lagerbestand.vorerfasst == true
Lagerbestand.vorerfasst == false
Lagerbestand.geloescht == true
Lagerbestand erfassen
Buchung ablehnen
Ablehnung protokollieren
Buchung.vorhanden == true
Buchung.vorhanden == false
subAktivität : Restwert verrechnen
Buchungswert bestimmen
Autorisierungsperson hinzufügen
Belege.uebrig == true
Belege.uebrig == false
Buchungen durchführen

C.2. SE-DSNL

This section contains additional material for the case study presented in Section 10.3.

C.2.1. Analysis Overview

The following analyses have been defined in the context of the SE-DSNL project.

Semantic Scope

Attribution	Meta Class	Description
semantic_symbol _references_element	SemanticSymbol	Checks whether all Semantic Symbols have a referencesElement value set
circle_generalizations	SemanticElement	An attribution that tests the Generalizations of SemanticElements for circles
semantic_element _in_hierarchy semantic_elements _in_hierarchy	SemanticElement Domain	Checks whether all SemanticElements belong to a single hierarchy

Syntactic Scope

Attribution	Meta Class	Description
syntactic_category _in_hierarchy syntactic_categories _in_hierarchy	SyntacticCategory Domain	Ensures SyntacticCategory hierarchy does not contain cycles
circle_syntacticcategories	SyntacticCategory	An attribution that tests the generalizations of SyntacticCategory for cycles
category_for_form	Form	Checks whether each Form or FormRoot has a SyntacticCategory set
formroot_for_form	Form	Checks whether each Form belongs to a FormRoot
syntactic_symbol _references_element	SyntacticSymbol	Checks whether all Syntactic/SemanticSymbols have a referencesElement value

Construction Scope

Attribution	Meta Class	Description
circle_constructions	Construction	Ensures Construction hierarchy does not contain cycles
construction_in_hierarchy	Construction Domain	Checks whether the Construction hierarchy is consistent
symbols_in_construction	Construction	Checks if a Construction contains all Symbols used by its referenced mappings

Interpretation Scope

Attribution	Meta Class	Description
interpretation_element _associated_to_model	InterpretationElement	Checks if InterpretationElements are associated with a model
semantic_element _interpretation _is_of_semantic_type	SemanticElement-Interpretation	Checks SemanticElementInterpretations for their semantic type
association_interpretation _is_of_type	AssociationInterpretation	Checks whether an AssociationInterpretation has a defined type
sem_int_element _for_association_int	AssociationInterpretation	Checks whether each AssociationInterpretation references a SemanticInterpretationElement via refersSemIntElem
ident_element_identified _elements_connected	IdentificationElement	When a IdentificationElement references an element through dentifiedBy, all referenced elements must form a connected subgraph
abstract_pattern _element_check	PatternElement PatternElement AbstractPatternElement	Performs sanity checks on PatternElements, e.g. association with Patterns

Helpers

Attribution	Meta Class	Description
all_con_successors	Construction	Calculates all successors (in upper direction) of a Construction
all_syncat_successors	SyntacticCategory	The attribution to calculate all (SyntacticCategory) successors for a node
all_gen_successors _objects	Element	The attribution to calculate all (Generalization) successors for a node
root_for_semantic _element	SemanticElement	Returns the name of the root element of a SemanticCategory
root_for_syntactic _category	Element	Returns the name of the root element of a SyntacticCategory
element_with_single _parent	Element	Checks whether an element has only one parent (of the same type)

C.2.2. Attribution construction_in_hierarchy

The following section contains the information omitted in Section 10.3.3 for the use case construction_in_hierarchy.

Attribution: construction_in_hierarchy

The attribution specification construction_in_hierarchy (cf. Algorithm 41) defines two attributes: The assignment constructions_in_hierarchy_assign for the element Domain computes a map that stores information about the validity of the Construction elements' roots while the constraint constructions_in_hierarchy_const uses this result (amongst others) to compute the final verdict for each Construction.

```
attribution construction_in_hierarchy {
    description "Checks if each Construction is part of a a hierarchy,
        not in a circle and if there exists only one root construction";

    attribute assignment constructions_in_hierarchy_assign
        : javaObject initWith hashmap;
    attribute constraint
        construction_in_hierarchy_const : error;

    rule java rule_constructions_in_hierarchy :
        call "brmsrules.constructions_in_hierarchy";
    rule java rule_construction_in_hierarchy :
        call "brmsrules.construction_in_hierarchy";

    extend Domain with {
        occurrenceOf constructions_in_hierarchy_assign
            calculateWith rule_constructions_in_hierarchy;
    }
    extend Construction with {
        occurrenceOf construction_in_hierarchy_const
            calculateWith rule_construction_in_hierarchy;
    }
}
```

Rule: root_for_element

The rule root_for_element computes for each Element the set of root Elements accessible through the Generalization hierarchy. This is done by evaluating the set of (transitive) parent Generalization elements through the successor relationship (attribute all_gen_successors_objects_assign). If no successor elements exist, the local Element is a root node and added to the result. Otherwise, the root_for_element value of the successors is recursively added to the result.

```
public Object root_for_element(IAttributeAccessor accessor, AttrDefinition definition, EObject localObject)
        throws InstantiatorException, VisualizerException, InvokerException, EvaluatorException {

    Set<EObject> successors = (Set<EObject>) accessor. getAttributeValueForObject(localObject,
                "all_gen_successors_objects_assign");

    Collection<EObject> roots = new HashSet<EObject>();
    if (successors.size() == 0)
        roots.add(localObject);
    else
        for (EObject succ : successors)
            if(!succ.equals(localObject)) {
                Set<EObject> succSuccessors = (Set<EObject>) accessor.getAttributeValueForObject(localObject,
                        "all_gen_successors_objects_assign");
                if(!succSuccessors.contains(localObject))
                    roots.addAll( (List<EObject>)accessor. getAttributeValueForObject(succ,
                            "root_for_element_assign"));
            }

    return roots;
}
```

Rule: constructions_in_hierarchy

The implementation of constructions_in_hierarchy first collects all Constructions contained in the Domain and requests the values of the root_for_element_assign attribute for all Constructions. This list of lists is merged into a set of unique names (uniqueRoots). The list of Constructions is then processes, adding a map entry to failedObjects with the Construction as key and

- a list of all global root nodes if the single inheritance property is violated
- an empty list if the root list is empty (indicating a cycle)

as value.

```
public Object constructions_in_hierarchy(IAttributeAccessor accessor, AttrDefinition definition, EObject localObject)
        throws InstantiatorException, VisualizerException, InvokerException, EvaluatorException {

    EList<EObject> elementList = accessor.collectEObjectsByClassName(accessor.getEReferenceValues(localObject, "contains"),
            new String[] { "Construction" });

    List<Object> rootLists = (List<Object>) accessor. getAttributeValuesForObjects(elementList, "root_for_element_assign");
    List<EObject> uniqueRoots = collapseRootNames(rootLists);

    Map<EObject, List<EObject>> failedObjects = new HashMap<EObject, List<EObject>>();
    for (EObject currentObject : elementList) {
        List<EObject> rootList = (List<EObject>) rootLists.get(elementList.indexOf(currentObject));
        if ((uniqueRoots.size() > 1 && uniqueRoots.contains(currentObject)))
            failedObjects.put(currentObject, uniqueRoots);
        else if (rootList.isEmpty())
            failedObjects.put(currentObject, rootList);
    }
    return failedObjects;
}
```

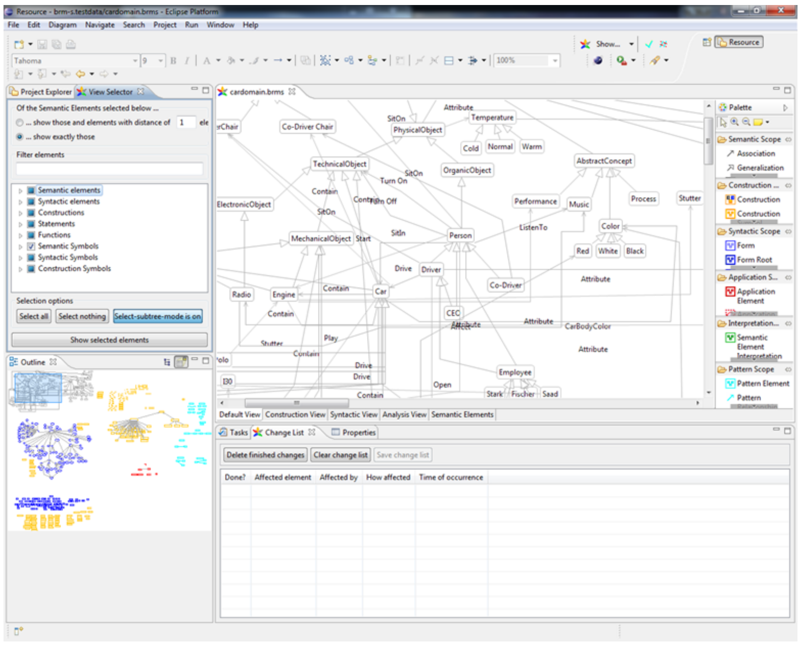
Rule: element_with_single_parent

The rule element_with_single_parent, defined for Element, queries all Generalization relationships and checks whether the references class type equals the class of the local object to ensure a consistent Generalization hierarchy. In addition, all parent Elements are collected so that constraints on the multiple inheritance property can be defined.

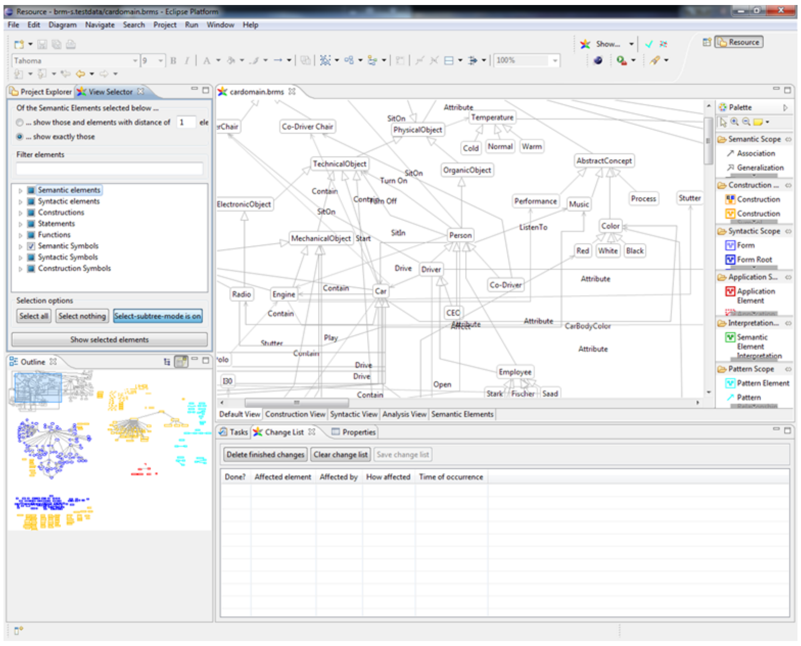
```
public Object element_with_single_parent(IAttributeAccessor accessor, AttrDefinition definition, EObject localObject)
        throws InstantiatorException, VisualizerException, InvokerException, EvaluatorException {

    String name =  EMFDynamicUtils.getObjectIdentification(localObject, null);

    EList<EObject> targetList = new BasicEList<EObject>();
    for (EObject elem :  accessor.getEReferenceValues(localObject, "contains")) {
        if (elem.eClass().getName().equals("Generalization")) {
            EObject target = accessor.getEReferenceValue(elem, "references");
            if (!target.eClass().equals(localObject.eClass())) {
                // no hierarchy when parent has a wrong type
                String message = "[" + localObject.eClass().getName() + "] \"" + name + "\" has parent with type ["
                        + target.eClass().getName() + "].";
                accessor.setConstraintMessage(localObject, definition.getId(), message);
            }
            targetList.add(target);
        }
    }

    return targetList;
}
```

C.2.3. SE-DSNL Editor

The graphical editor of the SE-DSNL DSL:



The construction editor:



Appendix D. Model Analysis Framework

D.1. Framework Core

Framework Instantiation

The class MAFCore provides several static methods for instantiating the framework.

Method	createFrameWork	
Description	Creates a new instance of the framework corresponding to the set of parameters. Loggers can be provided to record the initialization process.	
Parameter	**Type**	**Description**
parameters	CoreParameters	parameters for initializing the core
statusLoggers	Map<String, IVisualizerStatus>	initial set of status loggers
repositoryLoggers	Map<String, IVisualizerRepository>	initial set of repository loggers
Return type	IMAFCore	the instance of the framework

Method	createFrameWorkInteralAccess	
Description	Same as previous method. The returned instance of the type MAFCore allows additional access to internal functions.	
Parameter	**Type**	**Description**
parameters	CoreParameters	parameters for initializing the core
statusLoggers	Map<String, IVisualizerStatus>	initial set of status loggers
repositoryLoggers	Map<String, IVisualizerRepository>	initial set of repository loggers
Return type	MAFCore	the instance of the framework

Method	createFrameWorkInteralAccess	
Description	Same as previous method. Can additionally register debug loggers (combines status, repository, evaluation loggers and result processor).	
Parameter	**Type**	**Description**
parameters	CoreParameters	parameters for initializing the core
statusLoggers	Map<String, IVisualizerStatus>	initial set of status loggers
repositoryLoggers	Map<String, IVisualizerRepository>	initial set of repository loggers
debugLoggers	List<IVisualizerDebug>	the debug loggers
Return type	MAFCore	the instance of the framework

Module Access

The instance of MAFCore provides internal and external access to all modules of the framework:

Method	getRepositoryFacade	
Return type	RepositoryFacade	reference to the resource repository module (cf. Appendix D.4)

Method	getVisualizerFacade	
Return type	VisualizerFacade	reference to the logger/processor module (cf. Appendix D.5)

Method	getEvaluatorFacade	
Return type	EvaluatorFacade	reference to the strategy executor module (cf. Appendix D.7)

Additional Modules

The core also provides access to additional modules (mainly for internal use).

Method	getJavaClassLocator	
Return type	JavaClassLocator	MAF interface for locating and instantiating Java classes

Method	getThreadPoolExecutor	
Return type	MAFThreadPoolExecutor	MAF interface for scheduling tasks for processing using a parallelized working queue

Logging

The available log levels (relevant log level is defined in the set of core parameters):

Artifact type	**Description**
DEBUG	verbose debug information
INFO	status information
WARNING	non-critical error
ERROR	critical error (usually aborts execution)

Method	logStatus	
Description	Logs a status message using MAF's logging facilities. Registered status loggers are notified on each log entry.	
Parameter	**Type**	**Description**
source	Class<?>	the class which generates the log entry
level	int	log level
message	String	the log message
e	Exception	(optional) exception that should be logged

D.2. Artifact Adapters

Artifact adapters are used to load source artifacts into EMF resources. The adapters represent wrappers for the resources and can be stored in MAF's internal resource repository (cf. Appendix D.4). The actual loading process takes place once an adapter is stored in the repository.

Metamodel Adapter

Implemented in the MetaModelAdapter class.

Constructor		
Description	Constructs a new metamodel adapter by merging the provided input artifacts into a single resource.	
Parameter	**Type**	**Description**
metamodelSources	Map<Integer, Tuple<String, Object»	input resource map. the key represents the resource type while the value contains a reference to the artifact in the respective format (e.g. a file path).
metamodelID	String	the intended repository id for the adapter

Available artifact types for metamodels:

Artifact type	**Description**
resource	loads the metamodel from an EMF resource
ecore	loads the metamodel from an Ecore file using a URI path
generated	loads the metamodel using a EPackage class generated by EMF

Method	getMetaModelResourceMap	
Description	Returns a map of the resources representing the loaded artifacts.	
Return type	HashMap<URI, Resource>	a map relating the artifact URIs to the loaded resources

Method	getMetaModelInfo	
Return type	String	a list of the source artifacts from which this adapters contents were loaded

Method	getMetaModelRepoID	
Return type	String	returns the repository id of the adapter

Attribution Adapter

Implemented in the AttributionAdapter class.

Constructor		
Description	Constructs a new attribution adapter by merging the provided input artifacts into a single resource.	
Parameter	**Type**	**Description**
parameters	AttributionParameters	The parameter set used to configure the loading of the attribution
attributionSources	Map<URI, Object>	input resource map. the key represents the URI of the artifact while the value may contain additional data
attributionID	String	the intended repository id for the adapter
metamodelID	String	the repository id of the corresponding metamodel adapter

Method	getMetaModelResourceMap	
Description	Returns a map of the resources representing the loaded artifacts.	
Return type	HashMap<URI, Resource>	a map relating the artifact URIs to the loaded resources

Method	getAttributionInfo	
Return type	String	a list of the source artifacts from which this adapters contents were loaded

Method	getAttributionRepoID	
Return type	String	returns the repository id of the adapter

Method	getMetaModelRepoID	
Return type	String	returns the repository id of the corresponding metamodel adapter

Method	getMergedAttribution	
Return type	Set<String>	the list of the ids of the defined attributions

Method	getMergedAttribution	
Return type	Attribution	this object contains the loaded attribution consists of all source attributions that have been extended to represent the inheritance relationships

Model Adapter

Implemented in the ModelAdapter class.

Constructor		
Description	Constructs a new model adapter by merging the provided input artifacts into a single resource.	
Parameter	**Type**	**Description**
modelSources	Map<Integer, Tuple<String, Object»	input resource map. the key represents the resource type while the value contains a reference to the artifact in the respective format (e.g. a file path).
modelID	String	the intended repository id for the adapter
metamodelID	String	the intended repository id of the corresponding metamodel adapter

Available artifact types for models:

Artifact type	Description
resource	loads the model from an EMF resource
xmi	loads the model from an EMF XMI/Xtext file using a URI path

Method	getTraceMap	
Return type	String	returns a map that links internal EObjects to elements from the original source artifact

Method	getModelResourceMap	
Description	Returns a map of the resources representing the loaded artifacts.	
Return type	HashMap<URI, Resource>	a map relating the artifact URIs to the loaded resources

Method	getModelInfo	
Return type	String	a list of the source artifacts from which this adapters contents were loaded

Method	getModelRepoID	
Return type	String	returns the repository id of the adapter

D.3. Instantiations

An Instantiation is a wrapper for attribution that has been instantiated for a model, i.e. an attributed model. It provides the facilities for attribute instantiation and storage as well as the necessary functions for instance access and manipulation during the analysis. In the normal case, the presented methods are called internally by the framework. However, it is also possible to invoke them manually by retrieving the Instantiation object from MAF's repository.

Initialization

Creation and initialization of the Instantiation object is usually managed by the repository (cf. Appendix D.4).

Constructor		
Description	Creates a new Instantiation object representing the attributed model for the defined resources.	
Parameter	**Type**	**Description**
core	MAFCore	parent MAF Core instance.
parameters	InstantiationParameters	parameters configuring the Instantiation
metamodelID	String	repository id of the metamodel resource
modelID	String	repository id of the model resource
attributionID	String	repository id of the attribution resource

Method	initInstantiation	
Description	sets up the internal data structures and required functionality (invokers, accessor etc.)	
Parameter	**Type**	**Description**
useExistingModels	boolean	if true, existing data structures are reused (speeds up reinitialization in test mode)

Attribute Instantiation

The following methods provide support for the creation of attribute instances. They are called by the Strategy Executor to collect the selected instances.

Method	instantiateAttribute	
Description	Instantiates a specific attribute for an object	
Parameter	**Type**	**Description**
object	EObject	the object at which the attribute is annotated
attrDefinition	AttrDefinition	the attribute definition that should be instantiated
forceInitialization	boolean	force initialization with init rule
Return type	AttrInstance	the instantiated attribute

Method	instantiateAttributes	
Description	Instantiates all attributes for an object	
Parameter	**Type**	**Description**
object	EObject	the object at which attributes are annotated
forceInitialization	boolean	force initialization with init rule
Return type	Set<AttrInstance>	the instantiated attributes

Evaluation Management

The following methods manage the lifecycle of an Evaluation Target. They are mainly invoked by the Strategy Executor.

Method	evaluateInit
Description	Initializes the internal data structures for an evaluation run. Called by the Strategy Executor

Method	evaluateAttributes	
Description	Starts the evaluation for the set of given attribute instances. Called by the Strategy Executor	
Parameter	**Type**	**Description**
attrInstances	Collection<AttrInstance>	the instances to evaluate
Return type	Collection<AttrInstance>	the evaluated instances

Method	evaluateRecursiveCall	
Description	Handles recursive requests for attribute values from inside data-flow rules. Request is handed on to the selected evaluator's callback method. Called by the Attribute Accessor	
Parameter	**Type**	**Description**
attrInstance	AttrInstance	the requested instance

Method	evaluateFinish	
Description	Wraps up an evaluation of a target by generating statistical information and collecting the results. Called by the Strategy Executor	
Parameter	**Type**	**Description**
startTimeCompleteNS	long	start time of the strategy execution
startTimeEvaluationNS	long	start time of the target's evaluation
requestedResultInstances	Collection<AttrInstance>	the set of requested instances (without dynamically discovered instances)
allResultInstances	Collection<AttrInstance>	all instances (including discovered)
Return type	SimpleEvaluationResult	the results of the current Evaluation Target

Rule Invocation

Evaluators and the accessor can use the following methods to invoke data-flow rules:

Method	invokeRule	
Description	Invokes an attribute's initialization or iteration rule. Triggers callback if additional attribute values are requested	
Parameter	**Type**	**Description**
forceExecution	boolean	Can override the blockStable parameter
initializationRule	boolean	indicates whether the initialization rule should be executed
attrInstance	AttrInstance	the attribute instance which should be (re)computed
Return type	Object	result of the rule's execution

Method	calculateAttributeValue	
Description	Invokes an attribute's initialization or iteration rule without saving the result. Used e.g. by the AttributeAccessor's initialization mode where only initialization values are returned.	
Parameter	**Type**	**Description**
initializationRule	boolean	indicates whether the initialization rule should be executed
attrInstance	AttrInstance	the attribute instance which should be (re)computed
Return type	Object	result of the rule's execution

Data Structures

The following methods allow access to an Instantiation's internal data structures.

Method	getTraceMapCopyOrig	
Description	Trace map built by model adapter relating internal to original objects	
Return type	Map<EObject, Object>	trace map

Method	getTraceMapOrigCopy	
Description	Trace map built by model adapter relating original to internal objects	
Return type	Map<Object, EObject>	trace map

Method	getAttributeInstantiationModel	
Description	Allows access to the instantiated attribution	
Return type	AttributionInstantiation	the attributed model

Method	getStorageInstantiation	
Description	Static storage map that can be used by data-flow rules. Its lifecycle corresponds to the Instantiation's lifecycle	
Return type	Map<Object, Object>	storage map

Method	getStorageEvaluation	
Description	Static storage map that can be used by data-flow rules. Its lifecycle corresponds to an evaluation run	
Return type	Map<Object, Object>	storage map

Other Information

These methods provide access to additional information:

Method	getAttributeAccessor	
Description	Returns the AttributeAccessor of the Instantiation	
Return type	IAttributeAccessor	attribute accessor used by the Instantiation

Method	getParameters	
Description	Allows read/write access to the InstantiationParameters	
Return type	InstantiationParameters	parameters with which the Instantiation was created

Method	getMetaModelRepoID	
Description	Returns the id for the Instantiation's metamodel repository entry	
Return type	getMetaModelRepoID	metamodel repository id

Method	getModelRepoID	
Description	Returns the id for the Instantiation's model repository entry	
Return type	String	model repository id

Method	getAttributionRepoID	
Description	Returns the id for the Instantiation's attribution repository entry	
Return type	String	attribution repository id

D.4. Resource Repository

The resources are managed internally through adapters. The repository in which these adapters are stored can be accessed through the framework instance: mafCore.getRepositoryFacade(). There are overall four types of resources which are stored inside the repositories: Metamodels, attributions, models and instantiations.

Metamodel Repository

Methods for managing metamodel resources.

Method	loadMetaModel	
Description	Loads a metamodel adapter into the repository.	
Parameter	**Type**	**Description**
mmadapter	IMetaModelAdapter	metamodel adapter that should be loaded

Method	reloadMetaModel	
Description	Updates the metamodel (reloads the source artifact). May trigger reloading of depending resources.	
Parameter	**Type**	**Description**
metamodelID	String	repository id of the metamodel adapter to reload

Method	removeMetaModel	
Description	Removes the metamodel from the repository. May trigger removal of depending resources.	
Parameter	**Type**	**Description**
metamodelID	String	repository id of the metamodel adapter to remove

Method	lookupMetaModel	
Description	Retrieves a metamodel adapter based on its repository id.	
Parameter	**Type**	**Description**
metamodelID	String	repository id of the metamodel adapter to locate
Return type	MetaModelAdapter	the metamodel adapter

Method	iteratorMetaModel	
Description	Provides an iterator for all registered metamodel adapters.	
Return type	Iterator<MetaModelAdapter>	iterates over the loaded adapters

Attribution Repository

Methods for managing attribution resources.

Method	loadAttribution	
Description	Loads an attribution adapter into the repository.	
Parameter	**Type**	**Description**
aadapter	IAttributionAdapter	attribution adapter that should be loaded

Method	reloadAttribution	
Description	Updates the attribution (reloads the source artifact). May trigger reloading of depending resources.	
Parameter	**Type**	**Description**
attributionID	String	repository id of the attribution adapter to reload
metamodelID	String	repository id of the associated metamodel adapter

Method	removeAttribution	
Description	Removes the attribution from the repository. May trigger removal of depending resources.	
Parameter	**Type**	**Description**
attributionID	String	repository id of the attribution adapter to remove
metamodelID	String	repository id of the associated metamodel adapter

Method	lookupAttribution	
Description	Retrieves an attribution adapter based on its repository id.	
Parameter	**Type**	**Description**
attributionID	String	repository id of the attribution adapter to locate
metamodelID	String	repository id of the associated metamodel adapter
Return type	AttributionAdapter	the attribution adapter

Method	iteratorAttribution	
Description	Provides an iterator for all registered attribution adapters.	
Return type	Iterator<AttributionAdapter>	iterates over the loaded adapters

Model Repository

Methods for managing model resources.

Method	loadModel	
Description	Loads a model adapter into the repository.	
Parameter	**Type**	**Description**
madapter	IModelAdapter	model adapter that should be loaded

Method	reloadModel	
Description	Updates the model (reloads the source artifact). May trigger reloading of depending resources.	
Parameter	**Type**	**Description**
modelID	String	repository id of the model adapter to reload
metamodelID	String	repository id of the associated metamodel adapter

Method	removeModel	
Description	Removes the model from the repository. May trigger removal of depending resources.	
Parameter	**Type**	**Description**
modelID	String	repository id of the model adapter to remove
metamodelID	String	repository id of the associated metamodel adapter

Method	lookupModel	
Description	Retrieves a model adapter based on its repository id.	
Parameter	**Type**	**Description**
modelID	String	repository id of the model adapter to locate
metamodelID	String	repository id of the associated metamodel adapter
Return type	ModelAdapter	the model adapter

Method	iteratorModel	
Description	Provides an iterator for all registered model adapters.	
Return type	Iterator<ModelAdapter>	iterates over the loaded adapters

Instantiation Repository

Methods for managing Instantiations.

Method	instantiateAttribution	
Description	Instantiates an attribution for the selected resources. May trigger computation of the instances' results.	
Parameter	**Type**	**Description**
parameters	InstantiationParameters	The parameters set that configures the instantiation process
metamodelRepoID	String	the repository id of the metamodel adapter
modelRepoID	String	the repository id of the model adapter
attributionRepoID	String	the repository id of the attribution adapter

Method	reinstantiateAttribution	
Description	Resets the instantiation. May trigger computation of the instances' results.	
Parameter	**Type**	**Description**
metamodelRepoID	String	the repository id of the metamodel adapter
modelRepoID	String	the repository id of the model adapter
attributionRepoID	String	the repository id of the attribution adapter

Method	removeInstantiation	
Description	Removes the instantiation from the repository	
Parameter	**Type**	**Description**
metamodelRepoID	String	the repository id of the metamodel adapter
modelRepoID	String	the repository id of the model adapter
attributionRepoID	String	the repository id of the attribution adapter

Method	lookupInstantiation	
Description	Retrieves an instantiation based on the repository id of its resources.	
Parameter	**Type**	**Description**
metamodelRepoID	String	the repository id of the metamodel adapter
modelRepoID	String	the repository id of the model adapter
attributionRepoID	String	the repository id of the attribution adapter
Return type	AttributeInstantiationWrapper	the instantiation

Method	iteratorInstantiation	
Description	Provides an iterator for all instantiated attributions.	
Return type	Iterator<AttributeInstantiationWrapper>	iterates over the loaded instantiations

D.5. Loggers and Result Processors

Status, repository and evaluation loggers as well as Result Processors can be registered through the VisualizerFacade. They represent listeners which are notified on status messages, changes to the internal repositories, during and after the evaluation process. Through these concepts, one can access the status of the framework and the results of analyses (Status Loggers and Result Processors) which is useful when integrating MAF into existing applications. Evaluation and Repository Loggers are especially useful if one intends to implement custom IDE's for MAF. For convenience reasons, a debug logger concept is also provided that unifies all four types of listeners. For each type MAF includes standard implementations that e.g. write the received information to the console or into a text file. Internally, notifications are passed to the instance of the MAFCore class in the case of status messages and to the VisualizerFacade for all other types of logging entries.

Status Logger

The following methods have to be provided by status logger implementations:

Method	logStatus	
Description	Logs a standard status message	
Parameter	**Type**	**Description**
source	Class<?>	the class in which the message originated
level	CoreParametersLogLevel	the log level
debug	boolean	set to true if level is debug
currentTime	Date	the time at which the status message was generated
timeSpan	(optional) long	time span in nano seconds
message1	String	the message
message2	(optional) String	optional second part of message

Method	parametersChanged
Description	Notified on parameter change

Parameter	**Type**	**Description**
currentTime	Date	the time at which the status message was generated
parameters	IParameters	parameter set that has changed

Repository Logger

The repository logger messages can monitor changes to MAF's internal repositories.

Method	metaModelAdded
Description	Notified when a metamodel adapter has been successfully loaded into the repository

Parameter	**Type**	**Description**
currentTime	Date	the time at which the adapter was added
mmadapter	IMetaModelAdapter	the added metamodel adapter
repositoryIterator	Iterator<? extends IMetaModelAdapter>	iterator for metamodel adapters

Method	metaModelRemoved
Description	Notified when a metamodel adapter has been successfully removed from the repository

Parameter	**Type**	**Description**
currentTime	Date	the time at which the adapter was removed
mmadapter	IMetaModelAdapter	the removed metamodel adapter
repositoryIterator	Iterator<? extends IMetaModelAdapter>	iterator for metamodel adapters

Method	metaModelReloaded
Description	Notified when a metamodel adapter has been successfully reloaded

Parameter	**Type**	**Description**
currentTime	Date	the time at which the adapter was reloaded
mmadapter	IMetaModelAdapter	the reloaded metamodel adapter
repositoryIterator	Iterator<? extends IMetaModelAdapter>	iterator for metamodel adapters

Note: Repository loggers also contain similar methods for model and attribution adapters.

Method	visualizerAdded
Description	Notified when a visualizer has been added to the repository

Parameter	**Type**	**Description**
currentTime	Date	the time at which the visualizer has been added
id	String	the visualizer id
visualizer	IVisualizer	the visualizer

Method	visualizerRemoved
Description	Notified when a visualizer has been removed from the repository

Parameter	**Type**	**Description**
currentTime	Date	the time at which the visualizer has been removed
id	String	the visualizer id
visualizer	IVisualizer	the visualizer

Method	instantiationCreated
Description	Notified when a visualizer has been added to the repository

Parameter	**Type**	**Description**
iwrapper	IAttributeInstantiationWrapper	the Instantiation that has been created
repositoryIterator	Iterator<? extends IAttributeInstantiationWrapper>	iterator for Instantiations

Method	instantiationRemoved
Description	Notified when a visualizer has been removed from the repository

Parameter	**Type**	**Description**
iwrapper	IAttributeInstantiationWrapper	the Instantiation that has been removed
repositoryIterator	Iterator<? extends IAttributeInstantiationWrapper>	iterator for Instantiations

Evaluation Logger

Evaluation loggers enable tracking the state of the evaluation process and the solver's actions.

Method	startInstantiation
Description	Called when an Instantiation is (re)loaded

Parameter	**Type**	**Description**
currentTime	Date	the time on which the event occurred
iwrapper	IAttributeInstantiationWrapper	the (re)loaded Instantiation

Method	finishInstantiation
Description	Called when the initialization of an Instantiation is finished

Parameter	**Type**	**Description**
currentTime	Date	the time on which the event occurred
iwrapper	IAttributeInstantiationWrapper	the affected Instantiation

Method	startEvaluationDirective
Description	Called when an Evaluation Directive is executed

Parameter	**Type**	**Description**
currentTime	Date	the time on which the event occurred
evaluationDirective	IEvaluationDirective	the executed Evaluation Directive

Method	finishEvaluationDirective
Description	Called when the execution of an Evaluation Directive is finished

Parameter	**Type**	**Description**
currentTime	Date	the time on which the event occurred
evaluationDirective	IEvaluationDirective	the executed Evaluation Directive

Method	instantiatedAttribute
Description	Called when an Attribute Instance is created

Parameter	**Type**	**Description**
iwrapper	IAttributeInstantiationWrapper	the instance's containing Instantiation
instantiatedInstance	AttrInstance	the instantiated attribute

Method	initializedAttribute
Description	Called when an Attribute Instance is initialized

Parameter	**Type**	**Description**
iwrapper	IAttributeInstantiationWrapper	the instance's containing Instantiation
initializedInstance	AttrInstance	the initialized instance

Method	invokeRule
Description	Called when a data-flow rule is invoked

Parameter	**Type**	**Description**
iwrapper	IAttributeInstantiationWrapper	the instance's containing Instantiation
instance	AttrInstance	the affected instance
initialization	boolean	determines whether the initialization or iteration rule is invoked

Method	logMessage
Description	Logs a status message generated by the solver

Parameter	**Type**	**Description**
currentTime	Date	the time on which the event occurred
iwrapper	IAttributeInstantiationWrapper	the affected Instantiation
message	String	the log message

Method	logAttributes	
Description	Logs the current state of a list of instances	
Parameter	**Type**	**Description**
currentTime	Date	the time on which the event occurred
timeSpanNS	long	time span of the processing of the instances
iwrapper	IAttributeInstantiationWrapper	the affected Instantiation
evaluatedInstances	Collection<AttrInstance>	list of evaluated instances

Method	logDependencyChain	
Description	Logs the current state of a dependency chain	
Parameter	**Type**	**Description**
currentTime	Date	the time on which the event occurred
timeSpanNS	long	time span for the processing of the current chain
iwrapper	IAttributeInstantiationWrapper	the affected Instantiation
depChain	DependencyChain	the dependency chain model
depChainType	DependencyChainType	the type of the dependency chain
discoveredNodes	Collection<InstanceNode>	list of newly discovered instance nodes
discoveredEdges	Collection<Tuple<InstanceNode, InstanceNode»	list of newly discovered dependency edges
evaluatedNodes	Collection<InstanceNode>	nodes which have been evaluated in the current run

Result Processor

These methods are called during and after the evaluation process:

Method	processSimpleResult	
Description	Called with the result of a single Evaluation Target	
Parameter	**Type**	**Description**
simpleEvaluationResult	SimpleEvaluationResult	

Method	processAggregatedResult	
Description	Called with the aggregated result of a complete Evaluation Strategy	
Parameter	**Type**	**Description**
aggregatedEvaluationResult	AggregatedEvaluationResult	

D.6. Attribute Accessor

The Attribute Accessor can request attribute instance values from inside the data-flow rules. Additionally, it provides a number of helper functions that simplify the execution of model operations in Java code.

Attribute Access

The following methods can be used to access attribute instance values and manipulate constraint messages.

Method	getAttributeValueForObject	
Description	Requests a single attribute value and returns it	
Parameter	**Type**	**Description**
object	EObject	the object at which the attribute instance is located
attDefName	String	The attribute definition id
Return type	Object	iteration value of attribute instance

Method	getAttributeValuesForObject	
Description	Requests all attribute values for an object	
Parameter	**Type**	**Description**
object	EObject	the object at which the attribute instances are located
Return type	Map<String, Object>	iteration values of attribute instances

Method	getAttributeValuesForObjects	
Description	Requests attribute values at multiple objects for an attribute id	
Parameter	**Type**	**Description**
objects	EList<EObject>	the objects at which the attribute instances are located
attDefName	String	The attribute definition id
Return type	List<Object>	iteration value of attribute instance of the respective id at the given objects

Method	getAttributeValuesForObjects	
Description	Requests all attribute value at the given objects	
Parameter	**Type**	**Description**
objects	EList<EObject>	the objects at which the attribute instances are located
Return type	Map<String, Map<EObject, Object»	iteration values of all attribute instances at the given objects

Method	setConstraintMessage	
Description	Sets the information/error message for an attribute constraint	
Parameter	**Type**	**Description**
object	EObject	the objects at which the attribute instances are located
attDefName	String	the objects at which the attribute instances are located
message	String	the objects at which the attribute instances are located

Method	setInitializationMode	
Description	Puts the AttributeAccessor in initialization mode where each request results in the attribute's initialization value. Used by DFA solvers to build initial dependency relationships	
Parameter	**Type**	**Description**
initializationMode	boolean	initialization mode

Additional Evaluation Functions

These functions provide additional functionality that can be useful to implement DFA methods.

Method	getBitVectorUtil	
Description	Returns util for managing data in bitvectors	
Parameter	**Type**	**Description**
id	String	the id of a specific bitvector
Return type	BitVectorSetUtil	the bitvector util for the given id

Method	getStorageInstantiation	
Description	Returns a storage map that is specific to the current instantiation	
Return type	Map<Object, Object>	instantiation storage map

Method	getStorageEvaluation	
Description	Returns a storage map that is shared throughout an evaluation run	
Return type	Map<Object, Object>	evaluation storage map

Model Helper Functions

The following methods simplify model management in Java.

Method	getEClassName	
Description	Returns the name of an object's class	
Parameter	**Type**	**Description**
object	EObject	the EObject
Return type	String	class name

Method	collectEObjectsByClassName	
Description	Filters a list keeping only objects of given class names	
Parameter	**Type**	**Description**
eobjects	Collection<EObject>	the set of objects to filter
classNames	String[]	the filter class name(s)
Return type	EList<EObject>	the filtered list

Method	getEAttributeValue	
Description	Returns a class attribute value	
Parameter	**Type**	**Description**
object	EObject	the EObject
attName	String	the name of the class attribute
Return type	Object	class attribute value

Method	getEReferenceValue	
Description	Returns class reference value	
Parameter	**Type**	**Description**
object	EObject	the EObject
refName	String	the name of the class reference
Return type	Object	class reference value

Method	getEReferenceValues	
Description	Returns class reference values (for references with multiplicity greater than one)	
Parameter	**Type**	**Description**
object	EObject	the EObject
refName	String	the name of the class reference
Return type	Object	class reference values

Method	createUnion	
Description	Creates the union of multiple elements/sets	
Parameter	**Type**	**Description**
resolveContained	boolean	flattens the set(s)
objects	Object...	the elements/sets to unify
Return type	Set<?>	the unified set

Method	createIntersection	
Description	Creates the intersection of multiple sets	
Parameter	**Type**	**Description**
inputSets	Collection<?>	the sets to intersect
Return type	Set<?>	the intersection

D.7. Strategy Executor

The Strategy Executor is responsible for running an evaluation consisting of EvaluationDirectives. A directive is either an Evaluation Target or an Evaluation Macro. The executor can be accessed through the framework instance: mafCore.getEvaluatorFacade(). It exposes the following methods:

Method	evaluateTarget	
Description	Takes a single Evaluation Target and executes it	
Parameter	**Type**	**Description**
evaluationTarget	Evaluation Target	the target to execute
resultVisualizers	String[]	ids of registered result processors
Return type	SimpleEvaluationResult	results of the analysis

Method	evaluateDirectives	
Description	Takes a list of EvaluationDirectives and executes them	
Parameter	**Type**	**Description**
evaluationDirectives	List<IEvaluationDirective>	the directives to execute
resultVisualizers	String[]	ids of registered result processors
Return type	AggregatedEvaluationResult	aggregated results of the analysis

Method	evaluateDirectivesTest	
Description	Executes a list of EvaluationDirectives in test mode	
Parameter	**Type**	**Description**
testParameters	TestParameters	the parameters for the test run
evaluationDirectives	List<IEvaluationDirective>	the directives to execute
storageObjects	Map<Object, Object>	global storage accessible by data-flow rules
Return type	AggregatedEvaluationResult	aggregated results of the analysis

D.8. Project Set

The Project Set API allows the configuration of a MAF Core instance based on a ProjectSet model. The class exposing the necessary methods is called MAFProject-SetInterface.

Framework Initialization

Methods for initializing the framework using a Project Set configuration.

Method	createCore	
Description	Initialized MAF Core with the given Project Set model.	
Parameter	**Type**	**Description**
projectFileURI	URI	the URI path to the Project Set file
Return type	MAFCore	the initialized MAF Core instance

Method	createCoreWithDebugger	
Description	Same as createCore but additionally registers debug listeners.	
Parameter	**Type**	**Description**
projectFileURI	URI	the URI path to the Project Set file
debugVisualizers	List<IVisualizerDebug>	the debug visualizers that should be registered with the framework
Return type	MAFCore	the initialized MAF Core instance

Execution of Evaluation Strategies

Methods for triggering the execution of defined EvaluationStrategies.

Method	evaluateStrategy	
Description	Executes an Evaluation Strategy. The target model is located using the repository id of the strategy (the model has to be already present in the repository).	
Parameter	**Type**	**Description**
core	IMAFCore	the target framework core
strategyID	String	the id of the Evaluation Strategy
storageObjects	Map<Object, Object>	initial contents for the global storage maps which can be accessed by the data-flow rules
Return type	AggregatedEvaluationResult	the aggregated result of the strategy

Method	evaluateStrategy	
Description	Executes an Evaluation Strategy on the given model.	
Parameter	**Type**	**Description**
core	IMAFCore	the target framework core
modelAdapter	IModelAdapter	the adapter containing the target model
loadModelAdapter	boolean	indicates whether the artifacts should be loaded (otherwise its assumed, that they have already been loaded)
strategyID	String	the id of the Evaluation Strategy
storageObjects	Map<Object, Object>	initial contents for the global storage maps which can be accessed by the data-flow rules
Return type	AggregatedEvaluationResult	the aggregated result of the strategy

Execution of Evaluation Strategies in Test Model

Methods for triggering the execution of defined EvaluationStrategies in test mode.

Method	testStrategy
Description	Executes an Evaluation Strategy. The target model is located using the repository id of the strategy (the model has to be already present in the repository).

Parameter	**Type**	**Description**
core	IMAFCore	the target framework core
testParameters	TestParameters	the parameters configuring the test run
strategyID	String	the id of the Evaluation Strategy
storageObjects	Map<Object, Object>	initial contents for the global storage maps which can be accessed by the data-flow rules
Return type	AggregatedEvaluationResult	the aggregated result of the strategy

Method	testStrategy
Description	Executes an Evaluation Strategy on the given model.

Parameter	**Type**	**Description**
core	IMAFCore	the target framework core
testParameters	TestParameters	the parameters configuring the test run
modelAdapter	IModelAdapter	the adapter containing the target model
loadModelAdapter	boolean	indicates whether the artifacts should be loaded (otherwise its assumed, that they have already been loaded)
strategyID	String	the id of the Evaluation Strategy
storageObjects	Map<Object, Object>	initial contents for the global storage maps which can be accessed by the data-flow rules
Return type	AggregatedEvaluationResult	the aggregated result of the strategy

D.9. Project Set Grammar

Xtext grammar for the Project Set DSL:

```
ProjectSet returns projectset::ProjectSet:
 'projectset' id=ID (('version' version=STRING ';')? & ('description' description=STRING ';')?)
   ( coreParameters=CoreParameters? &
     ( 'logger' statusLoggers+=JavaLink ';')* &
     repoMetamodels+=MetaModelEntry* &
     repoAttributions+=AttributionEntry* &
     repoModels+=ModelEntry* &
     evaluationStrategies+=EvaluationStrategy* ) ;


CoreParameters returns parameters::CoreParameters:
 'core_parameters' '{'
   (('autodispose_policy' '=' autodisposePolicy=CoreParametersAutodisposePolicy ';')? &
   ('thread_count' '=' threadCount=INT ';')? &
   ('synchronize_resources' '=' synchronizeResources=EBoolean ';')? &
   ('log_level' '=' logLevel=CoreParametersLogLevel ';')? &
   ('eclipse_console_log' '=' eclipseConsoleLog=EBoolean ';')? &
   ('parameters_autoreset' '=' parametersAutoreset=EBoolean ';')?) '}' ;

enum CoreParametersAutodisposePolicy returns parameters::CoreParametersAutodisposePolicy:
 off='off' | adapters='adapters' | visualizers='visualizers' | both='both';

enum CoreParametersLogLevel returns parameters::CoreParametersLogLevel:
 all='all' | debug='debug' | info='info' | warning='warning' | error='error' | off='off';

MetaModelEntry returns repository::MetaModelEntry:
 'load_metamodel' repoID=ID '{' (resources+=InputResourceMetaModel)* '}';

InputResourceMetaModel returns repository::InputResource:
 ((type=InputArtifactTypeMetaModelEcore 'from' uri=STRING) |
  (type=InputArtifactTypeMetaModelGenerated javaLink=JavaLink)) ';' ;

enum InputArtifactTypeMetaModelEcore returns repository::InputArtifactType:
 ecore='ecore';

enum InputArtifactTypeMetaModelGenerated returns repository::InputArtifactType:
 generated='generated';

ModelEntry returns repository::ModelEntry:
 'load_model' repoID=ID ':' metamodel=[repository::MetaModelEntry] '{'
   (resources+=InputResourceModel)* '}' ;

InputResourceModel returns repository::InputResource:
 type=InputArtifactTypeModel 'from' uri=STRING ';' ;

enum InputArtifactTypeModel returns repository::InputArtifactType:
 xmi='xmi' ;

AttributionEntry returns repository::AttributionEntry:
 'load_attribution' repoID=ID ':' metamodel=[repository::MetaModelEntry]'{'
   (parameters=AttributionParameters? & (resources+=InputResourceAttributionEntry)*) '}' ;

InputResourceAttributionEntry returns repository::InputResource:
 type=InputArtifactTypeAttributionEntry 'from' importURI=STRING ';' ;

enum InputArtifactTypeAttributionEntry returns repository::InputArtifactType:
 xmi='attrmm' ;

AttributionParameters returns parameters::AttributionParameters:
 'parameters' '{'
   (('inheritance_policy' '=' inheritancePolicy=AttributionParametersInheritancePolicy ';')? &
    ('mock_java_rules' '=' mockJavaRules=EBoolean ';')? &
    ('static_rule_creation' '=' staticRuleCreation=EBoolean ';')? &
    ('javarules' javaRuleClasses+=JavaLinkWithID ';')*) '}' ;

enum AttributionParametersInheritancePolicy returns parameters::AttributionParametersInheritancePolicy:
 off='off' | inherit='inherit' | redefine='redefine' ;

EBoolean returns ecore::EBoolean:
 'true' | 'false' ;

JavaLink returns projectset::JavaLink:
 (('classpath' javaClassPath=STRING) | ('link' jvmLink= [jvmTypes::JvmType|QualifiedName])) ;

JavaLinkWithID returns projectset::JavaLink:
 id=ID (('classpath' javaClassPath=STRING) | ('link' jvmLink= [jvmTypes::JvmType|QualifiedName])) ;

EvaluationStrategy returns evaluation::EvaluationStrategy:
 'strategy' id=ID '{' (('description' description=STRING ';')? &
   ('resultprocessor' resultProcessors+=JavaLink ';')* &
   (instantiationParameters=InstantiationParameters)? &
   (directives+=EvaluationDirective)* ) '}' ;

InstantiationParameters returns parameters::InstantiationParameters:
 'parameters' '{'
   (('autodelete_policy' '=' autoDeletePolicy=InstantiationParametersAutodeletePolicy ';')? &
    ('block_stable' '=' blockStable=EBoolean ';')? &
    ('evaluator' '=' evaluatorType=InstantiationParametersEvaluatorType ';')? &
    ('evaluator_parameter' evaluatorParameters+=EvaluatorParameterEntry ';')* &
    ('measure_performance' '=' measurePerformance=EBoolean ';')? &
    ('validate_references' '=' validateReferences=EBoolean ';')? &
    ('debug_mode' '=' evaluatorDebugMode=EBoolean ';')? &
    ('synchronize_evaluation' '=' synchronizeEvaluation=EBoolean ';')? &
    ('instantiation_policy' '=' instantiationPolicy=InstantiationParametersInstantiationPolicy ';')? &
    ('problemmarker_policy' '=' problemMarkerPolicy=InstantiationParametersProblemmarkerPolicy ';')? &
    ('max_rule_invoke' '=' maxRuleInvoke=LONG ';')? ) '}' ;

EvaluatorParameterEntry returns projectset::StringToStringMapEntry:
 key=ID '=' value=STRING ;

enum EvaluatorParameterKey returns parameters::EvaluatorParameters:
 eval_depchain_bu_eliminate ='eval_depchain_bu_eliminate' |
 eval_depchain_bu_postpone='eval_depchain_bu_postpone' |
 eval_depchain_bu_start='eval_depchain_bu_start' | eval_depchain_phase1 ='eval_depchain_phase1' |
 eval_depchain_phase1_parallelize='eval_depchain_phase1_parallelize' |
 eval_depchain_phase2='eval_depchain_phase2' |
 eval_depchain_phase2_parallelize='eval_depchain_phase2_parallelize' |
 eval_depchain_wl_adddiscovered='eval_depchain_wl_adddiscovered' |
 eval_roundrobin_reclookup='eval_roundrobin_reclookup' | eval_worklist_changesets='eval_worklist_changesets' |
 eval_worklist_reclookup='eval_worklist_reclookup';

enum InstantiationParametersAutodeletePolicy returns parameters::InstantiationParametersAutodeletePolicy:
 off='off' | instantiation='instantiation' | all='all' ;

enum InstantiationParametersProblemmarkerPolicy returns parameters::InstantiationParametersProblemmarkerPolicy:
 off='off' | add='add' | replace='replace' ;

enum InstantiationParametersInstantiationPolicy returns parameters::InstantiationParametersInstantiationPolicy:
 static='static' | staticinit='staticinit' | ondemand='ondemand' ;

enum InstantiationParametersEvaluatorType returns parameters::InstantiationParametersEvaluatorType:
 depchain='depchain' | worklist='worklist' | roundrobin='roundrobin' ;

EvaluationDirective returns evaluation::EvaluationDirective:
 EvaluationTarget | EvaluationMacro ;

EvaluationTarget returns evaluation::EvaluationTarget:
 'target' '(' metamodel=[repository::MetaModelEntry] ','
   attribution=[repository::AttributionEntry] (',' model=[repository::ModelEntry])? ')'
   ('attributions' targetAttributions+=[attribution::Attribution]
      (',' targetAttributions+=[attribution::Attribution])* |
    'attributes' targetAttributes+=[attributes::AttrDefinition]
      (',' targetAttributes+=[attributes::AttrDefinition])* |
    'occurrences' targetOccurrences+=[attributes::AttrOccurrence]
      (',' targetOccurrences+=[attributes::AttrOccurrence])* ) ';' ;

EvaluationMacro returns evaluation::EvaluationMacro:
 'macro' id=ID javaLink=JavaLink ';' ;
```