Applying Data-flow Analysis to Models A Novel Approach for Model Analysis

Christian Saad, Bernhard Bauer
Programming Distributed Systems Lab, University of Augsburg, Germany
{saad, bauer}@informatik.uni-augsburg.de

Keywords: data-flow analysis, semantic analysis, MOF, meta modeling, business processes

Abstract

Using modeling techniques, the structure of an application domain can be captured in an easy and highly expressive way. However, while the use of meta models for the definition of modeling languages is a common and well-understood activity, extracting information about behavioral properties as well as the validation of static semantics is still a challenge.
In this paper we present a novel approach for model analysis that addresses these issues by applying the method of data-flow analysis to the modeling domain. By approximating the dynamic behavior of a model, this allows for an abstract interpretation of its runtime characteristics.

1. INTRODUCTION AND MOTIVATION

Today, meta modeling is a well established method to formally describe the structure of an application domain, e.g. the internal layout of software systems. This is facilitated by standards like the widely supported Unified Modeling Language (UML) or the Meta-Object Facility (MOF).

The importance of models raises the question of how to validate their correctness, an issue that is currently not solved satisfactorily by the UML and similar modeling languages. Aside from the abstract syntax given by a meta model, there are often additional restrictions known as static semantics which cannot be expressed using syntactical expressions alone. The Object Constraint Language (OCL) is intended to enable the definition of such well-formedness rules but due to its static nature, it is not capable of validating dynamic properties that are highly dependent on the context in which the elements appear, e.g. correct nesting of parallel paths in activities. Extracting knowledge about the dynamic properties would also allow to perform an abstract interpretation of a model, e.g. to determine valid execution paths in a workflow.

The approach presented in this paper overcomes the limitations of purely static methods like OCL by performing a dynamic flow analysis on models, thus offering a powerful and generically applicable method for model validation and simulation. Its basis is the well-understood data-flow analysis (DFA) technique used commonly in compiler construction to derive optimizations from a program's control flow graph.

In this paper we present an extended and updated definition of the concept of using DFA for model analysis introduced in [1] along with use cases to demonstrate its applicability. The paper is structured as follows: The principles of model-based data-flow analysis, its definition and an evaluation algorithm, are described in Section 2.. In Section 3., several use cases are presented which have already been implemented or are currently under evaluation. Section 4. contains an overview of related work, before we give a summary of the concepts described in this paper and an outlook on future developments.

2. APPLYING DATA-FLOW ANALYSIS TO MODELS

In [1], we have shown that data-flow techniques, stemming from the field of compiler construction, can be adopted for the modeling domain since both areas share the underlying principle of using multiple abstraction layers to define programming and modeling languages respectively.

To stay consistent with the notion that everything is a model, it is desirable that the flow equations themselves are defined through means of a meta model (described in section 2.1.). The definition is inspired by attribute grammars, a technique for static analysis of syntax trees in which the grammatical symbols are annotated with semantic attributes. While attribute grammars in their original form are too restrictive and difficult to integrate into the modeling domain, the basic concept forms a valid basis for enriching meta models with data-flow definitions which are therefore called attributions.

One of the major differences in comparison to the original DFA approach is that the flow graph which describes the actual data flow is not known beforehand but created during the execution. Data-flow equations may access semantic attributes located at arbitrary model elements, thereby superimposing an input dependency graph on the model's structure. The evaluation algorithm presented in section 2.2. is able to handle these dynamic dependency relationships in a way that significantly reduces the amount of execution steps, comparable to the worklist algorithm commonly employed for evaluating traditional DFA in compiler construction.

2.1. A Meta Model For Data-Flow Analysis

In accordance to the procedure commonly employed for attribute grammars, attributes are given a data type and an initial value. These attribute definitions can then be assigned to meta model classes (the M2 layer in the MOF terminology) by creating occurrences which link the attribute to the class

https://doi.org/10.1145/1878537.1878788

as well as to a semantic rule (the data-flow equation) which is responsible for calculating the attribute's instance layer result.

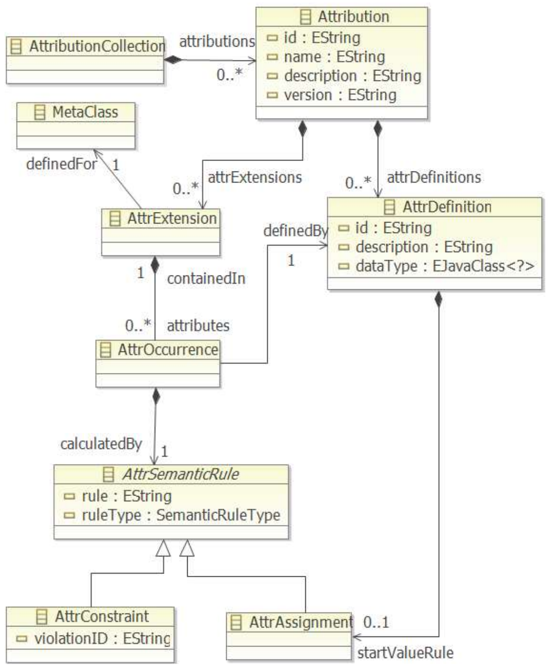


Figure 1. Refined attribution meta model (AttrMM)

The complete attribution meta model *AttrMM* is shown in Figure 1: *Attributions* are contained in the top-level element *AttributionCollection* and consist of a set of *AttributeDefinitions* and *AttributeExtensions*. *AttributeDefinitions* are given an identifier *id* along with a *dataType* and a single *AttrSemanticRule* of the type *AttrAssignment* which is responsible for returning the initialization value for this attribute. An *AttributeExtension* serves as a container for linking multiple occurrences of defined attributes (*AttrOccurrences*) to a target meta model class. Each *AttrOccurrence* contains exactly one semantic rule which calculates the iteration value for instances of this occurrence. Semantic rules can be of the type *AttrAssignment* which returns a result value of the specified *dataType* or *AttrConstraint* which evaluates to a boolean. In the latter case, "false" indicates an error in the model identified by the given *violationID*.

The attribution meta model is designed to only introduce dependencies in one direction thus allowing the target meta model to remain unaware of any attributes assigned to its classes. This is an important feature since most existing tools and algorithms (such as model transformations) do not tolerate modifications of the meta language, e.g. the integration of language elements for attributes into MOF. It also simplifies storage and versioning of attributions.

In order to evaluate an attribution for a model the defined attributes have to be instantiated with respect to the meta model's generalization hierarchy. This is accomplished by collecting all *AttributeExtensions* connected to the specific class type and the super types of a model element. An attribute instance (with a slot to hold the evaluation result) is then created for each *AttrOccurrence* contained in one of the *AttributeExtensions*. This inheritance semantics ensures that any attribute connected to class C is implicitly available at model elements which are instances of subclasses of C. Using the most specific occurrence provides support for redefinition at subtypes - a common feature in the domain of modeling - i.e. if two *AttrOccurrences* O_{C1} and O_{C2} of the same *AttributeDefinition* O were assigned to classes $C1$ and $C2$ and $C2$ is a subclass of $C1$, then O_{C2} overrides O_{C1} at all instances of $C2$.

Unlike our previous definition in [1], the input relationships between attribute occurrences are not explicitly modeled anymore since this not only complicates the definition of an attribution but may also introduce unnecessary dependencies on the instance layer. Therefore, the evaluation algorithm presented in the next section was extended to analyze dependencies dynamically during execution of the semantic rules.

2.2. Attribution Evaluation

The evaluation algorithm is responsible for executing the rules in a valid order ensuring that required input arguments are available at the time of execution. In the case of cyclic dependencies, it may be necessary to (re)evaluate attribute instances until results have become available at all participant objects and a stable set of final values (fix-point) is reached. The worklist algorithm, commonly employed for solving DFA equations (cf. [2]), is not applicable here since it depends on knowing the output relationships in order to update the set of depending variables after each execution.

Since in our case dependencies are not available until the execution of a rule, another approach was chosen: From the set of attribute instances to be evaluated, an instance is chosen nondeterministically and its associated semantic rule is invoked. If another attribute instance is requested as input, this dependency is recorded and the corresponding rule is invoked recursively. Cyclic dependencies (back edges) are replaced by virtual nodes which derive their value from the *AttrDefinition*'s init rule. This results in a directed acyclic graph with designated root and leaf nodes - referred to as dependency chain - representing the input/output relationships between the attribute instances. The process is repeated until all attributes in the evaluation set are part of a dependency chain.

After completing the initial building phase, values at virtual nodes are compared against their referenced nodes. If they differ, the virtual node is updated with the value at its reference node and the corresponding branch is reevaluated in a bottom-up fashion until all values are stable. Note that dependency chains may change during the evaluation phase if new dependencies are introduced which were hitherto concealed, e.g. by an "if" clause in the semantic rule.

This algorithms greatly reduces the amount of required invocations although there are possibilities for optimization, e.g. through isolated computation of cycles or parallelization.

2.3. Model Analysis Framework

To verify the feasibility of this approach the Model Analysis Framework (MAF, `http://code.google.com/p/model-analysis-framework/`) project was created. It serves as a highly parametrizable and modular research platform for improving the related definitions and algorithms as well as allowing to do performance tests under realistic settings (but is also suited for productive use). All involved (meta) models, including the internal *AttrMM* meta model, are handled using the Eclipse Modeling Framework (EMF). Support for other modeling standards can be implemented through an adapter interface.

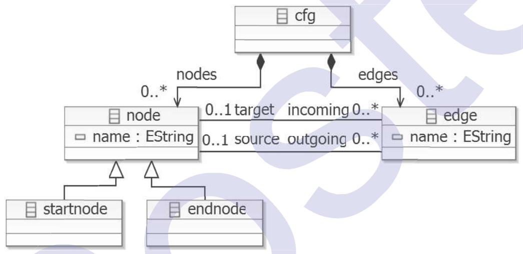
Semantic rules can be defined either in an enhanced version of either OCL or Imperative OCL (based on Eclipse OCL and Eclipse M2M) or Java. In both cases, attribute values located at other model elements can be requested through helper methods which trigger the evaluator module of the framework whose responsibility is to satisfy this dependency.

In the following section, an example use case that has been realized using MAF can be seen.

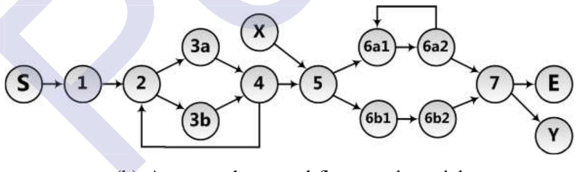
3. USE CASES

3.1. Analysis of Control-Flow Graphs

In this section we will show how to determine some basic properties for control-flow graphs (CFG) using a simple CFG meta model (cf. Figure 2(a)) along with a corresponding instantiated model (cf. Figure 2(b)).



(a) A simple control-flow graph meta model



(b) An example control-flow graph model

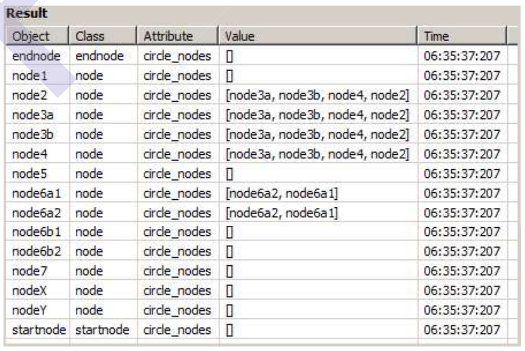
Figure 2. Control-flow graph example

The analysis is performed by the following attributes assigned to the class *node* and written in an extended OCL syntax which allows to request values of semantic attributes:

```
is_reachable:  self.incoming.source.is_reachable()→includes(true)

is_live:  self.outgoing.target.is_live()→includes(true)

all_predecessors:  self.incoming.source.name→
        union(self.incoming.source.all_predecessors())→asSet()

circle_id:  let self_pred : Set(String) =
        self.all_predecessors()→including(self.name) in
        if (self.incoming.source.all_predecessors()→asSet()=self_pred)
        then self_pred→hashCode() else 0 endif)

circle_nodes:  if (not(self.circle_id() = 0))
        then self.incoming.source→collect(predNode : node |
        if (predNode.circle_id()=self.circle_id())
        then predNode.circle_nodes() else Set{} endif)
        →flatten()→asSet()→including(self.name)
        else Set{} endif
```

The value of the constraint *is_reachable* indicates whether the associated node lies on a direct path from the start node ("false" for X). The start node itself overwrites this rule to always return "true". Accordingly, *is_live* checks the liveness, i.e. if the end node can be reached ("false" for Y).

To calculate the transitive predecessor set, *all_predecessors* merges the values of incoming *all_predecessors* attributes and adds the names of the direct predecessor nodes.



Result

Object	Class	Attribute	Value	Time
endnode	endnode	circle_nodes	[]	06:35:37:207
node1	node	circle_nodes	[]	06:35:37:207
node2	node	circle_nodes	[node3a, node3b, node4, node2]	06:35:37:207
node3a	node	circle_nodes	[node3a, node3b, node4, node2]	06:35:37:207
node3b	node	circle_nodes	[node3a, node3b, node4, node2]	06:35:37:207
node4	node	circle_nodes	[node3a, node3b, node4, node2]	06:35:37:207
node5	node	circle_nodes	[]	06:35:37:207
node6a1	node	circle_nodes	[node6a2, node6a1]	06:35:37:207
node6a2	node	circle_nodes	[node6a2, node6a1]	06:35:37:207
node6b1	node	circle_nodes	[]	06:35:37:207
node6b2	node	circle_nodes	[]	06:35:37:207
node7	node	circle_nodes	[]	06:35:37:207
nodeX	node	circle_nodes	[]	06:35:37:207
nodeY	node	circle_nodes	[]	06:35:37:207
startnode	startnode	circle_nodes	[]	06:35:37:207

Figure 3. Evaluation result: Nodes that are part of cycles

The attribute *circle_id* compares *all_predecessors* values at preceding nodes to the value at the local node in order to identify cycles in the CFG. If the sets are identical, *circle_id* returns an ID calculated from the cyclic elements' hash codes.

Finally, *circle_nodes* accesses *circle_id* to determine the nodes which are part of detected cyclic flows by creating a set of predecessor nodes with identical *circle_id*. The final values for this attribute can be seen in Figure 3.

Calculating *circle_nodes* with all dependencies requires 220-310 rule executions using an unoptimized evaluation algorithm and ca. 100 executions using the dependency chain method. Once the OCL environment has been initialized, overall execution takes about 50ms on a standard desktop computer. Implementing the rules in Java leads to more verbose definitions but reduces the time required to about 30ms.

3.2. Business Process Analysis

A common requirement for many algorithms dealing with business processes is the decomposition of the process graph into a hierarchical representation of single-entry-single-exit (SESE) components.

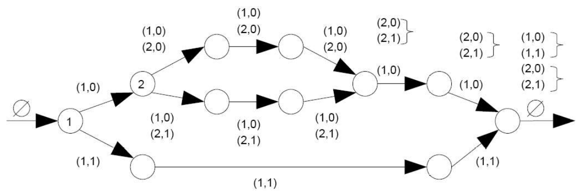


Figure 4. Decomposition of (business) processes [3]

The authors of [3] describe an algorithm based on token flow analysis, i.e. tokens which are created and merged at gateways and propagated along the flow direction as can be seen in Figure 3.2.. Tokens originating from the same node converge and are removed (indicated by curly brackets). SESE components can be identified by similar token labelings (determining the substructure of cycles requires some additional handling).

The creation, propagation and merging of tokens can be easily realized (65 lines of Java code in two rules and reuse of the attributes defined in Section 3.2.) using the generic DFA approach as opposed to a proprietary implementation.

Using the method described in [4], the soundness of a business process (i.e. the absence of local deadlocks and lack of synchronization) can be validated in linear time. This is achieved by traversing the resulting SESE tree and applying heuristics to categorize each component according to its internal structure. It is possible to already perform this evaluation during the process decomposition by integrating it into the DFA implementation of the algorithm presented above.

SESE decomposition also enables the transformation of graph-oriented BPMN diagrams (Business Process Modeling Notation) to block-oriented BPEL code (Business Process Execution Language). This is accomplished by performing a DFA on each recognized component as described in [5] resulting in a corresponding BPEL mapping. Since the algorithm is defined in the form of a data-flow analysis, its implementation using model-based DFA is straightforward.

4. CONCLUSIONS AND FUTURE INVESTIGATIONS

In this paper we have shown how the well-known method of data-flow analysis can be adopted for the modeling domain building upon widely accepted standards like OMG's Meta-Object Facility and the Object Constraint Language.

To the best of our knowledge, there is currently no comparable methodology which implements a generic DFA-oriented approach for the analysis of models, although data-flow techniques have been used in the modeling domain: Aside from the examples given in Section 3.2. it was shown in [6] that flow-analysis can be used to derive definition/use relationships between actions in state machines. A related technique can also be found in the generation of instance snapshots from meta models as a basis for validation ([7]).

In contrast to static techniques like OCL, flow-analysis allows to analyze (cyclic) information flows in the model graph based on local propagation and (re)calculation of attribute values. Aside from validation scenarios, extracting context-sensitive data enables to analyze dynamic aspects of models, e.g. valid execution paths in control-flows or the SESE components that make up a business process definition. Therefore, model-based DFA constitutes a generic and versatile "programming-language" for a wide variety of algorithms that would otherwise each require a proprietary definition.

The next steps include the exploration of new application areas, e.g. the extraction of metrics in the area of model-driven testing and the avoidance of complex OCL constraints to minimize the impact of meta model refactoring. Also, the performance of the evaluation algorithm will be improved.

5. REFERENCES

[1] Saad, C., F. Lautenbacher, and B. Bauer, 2009, "An Attribute-based Approach to the Analysis of Model Characteristics". *Proceedings of the First International Workshop on Future Trends of Model-Driven Development in the context of ICEIS'09.*

[2] Aho, A. V., M. S. Lam, R. Sethi, and J. D. Ullman, 2006, *Compilers – Principles, Techniques, & Tools.* Addison Weasley, 2nd edition.

[3] Götz, M., S. Roser, F. Lautenbacher, and B. Bauer, September 2009, "Token Analysis of Graph-Oriented Process Models". *New Zealand Second International Workshop on Dynamic and Declarative Business Processes (DDBP), in conjunction with the 13th IEEE International EDOC Conference (EDOC 2009).*

[4] Vanhatalo, J., H. Völzer, and F. Leymann, 2007, "Faster and More Focused Control-Flow Analysis for Business Process Models Through SESE Decomposition". In *ICSOC '07: Proceedings of the 5th international conference on Service-Oriented Computing*, Springer-Verlag, Berlin, Heidelberg, 43–55.

[5] García-Bañuelos, L., 2008, "Pattern Identification and Classification in the Translation from BPMN to BPEL". *OTM 08: Proceedings of the OTM 2008 Confederated International Conferences, CoopIS, DOA, GADA, IS, and ODBASE 2008* 436–444.

[6] Waheed, T., M. Iqbal, and Z. Malik, 2008, "Data Flow Analysis of UML Action Semantics for Executable Models". *Lecture Notes in Computer Science*, 5095, 79–93.

[7] Gogolla, M. and M. Richters, 2003, "Validation of UML and OCL Models by Automatic Snapshot Generation". In *Proceedings of the 6th Int. Conf. Unified Modeling Language*, Springer, 265–279.