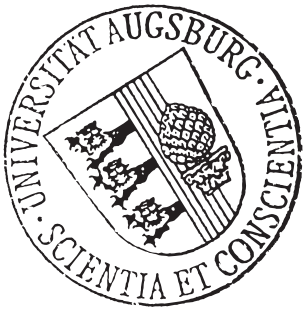
Towards a Model-Centric Software Testing Life Cycle for Early and Consistent Testing Activities

Reinhard Pröll

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



University of Augsburg
Department of Computer Science
Software Methodologies for Distributed Systems

April 2021

Software Methodologies
for Distributed Systems

Towards a Model-Centric Software Testing Life Cycle
for Early and Consistent Testing Activities

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

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

Thesis defense: 19th July 2021

Abstract

The constant improvement of the available computing power nowadays enables the accomplishment of more and more complex tasks. The resulting implicit increase in the complexity of hardware and software solutions for realizing the desired functionality requires a constant improvement of the development methods used. On the one hand over the last decades the percentage of agile development practices, as well as test-driven development increases. On the other hand, this trend results in the need to reduce the complexity with suitable methods. At this point, the concept of abstraction comes into play, which manifests itself in model-based approaches such as MDSD or MBT.

The thesis is motivated by the fact that the earliest possible detection and elimination of faults has a significant influence on product costs. Therefore, a holistic approach is developed in the context of model-driven development, which allows applying testing already in early phases and especially on the model artifacts, i.e. it provides a *shift left* of the testing activities. To comprehensively address the complexity problem, a model-centric software testing life cycle is developed that maps the process steps and artifacts of classical testing to the model-level.

Therefore, the conceptual basis is first created by putting the available model artifacts of all domains into context. In particular, structural mappings are specified across the included domain-specific model artifacts to establish a sufficient basis for all the process steps of the life cycle. Besides, a flexible metamodel including operational semantics is developed, which enables experts to carry out an abstract test execution on the model-level.

Based on this, approaches for test case management, automated test case generation, evaluation of test cases, and quality verification of test cases are developed. In the context of test case management, a mechanism is realized that enables the selection, prioritization, and reduction of Test Model artifacts usable for test case generation. I.e. a targeted set of test cases is generated satisfying quality criteria like coverage at the model-level. These quality requirements are accomplished by using a mutation-based analysis of the identified test cases, which builds on the model basis. As the last step of the model-centered software testing life cycle two approaches are presented, allowing an abstract execution of the test cases in the model context through structural analysis and a form of model interpretation concerning data flow information. All the approaches for accomplishing the problem are placed in the context of related work, as well as examined for their feasibility by of a prototypical implementation within the *Architecture And Analysis Framework*. Subsequently, the described approaches and their concepts are evaluated by qualitative as well as quantitative evaluation. Moreover, case studies show the practical applicability of the approach.

Zusammenfassung

Die stetige Verbesserung der verfügbaren Rechenleistung ermöglicht heutzutage das Bewältigen immer komplexerer Aufgaben. Die dadurch implizit steigende Komplexität der Hardware- aber auch Software-seitigen Lösungen zur Realisierung der gewünschten Funktionalität bedingt ebenfalls eine stetige Weiterentwicklung der hierfür genutzten Entwicklungsmethoden. Einerseits steigt über die letzten Jahrzehnte der Anteil an agilen Entwicklungspraktiken, aber auch Praktiken, die dem Testen der Funktionalität einen höheren Stellwert geben, in der Praxis deutlich an. Andererseits ergibt sich aus dieser Entwicklung die Notwendigkeit der Komplexität während der Entwicklungsarbeit durch geeignete Methoden zu begegnen. An dieser Stelle kommt oftmals das Konzept der Abstraktion zum Einsatz, was sich konkret in Modell-basierten Ansätzen wie MDSD oder MBT manifestiert.

Die vorliegende Arbeit ist motiviert durch die Erkenntnis, dass ein möglichst frühes Erkennen und Beheben von Fehlern deutlichen Einfluss auf die Produktkosten hat. Daher wird im Rahmen dieser Arbeit ein ganzheitliches Konzept entwickelt, welches im Kontext Modell-getriebener Entwicklung eine Möglichkeit eröffnet, Testen bereits in frühen Phasen und insbesondere auf den Modellartefakten anzuwenden, also einen *Shift Left* der Tests vorsieht. Um der Komplexitätsproblematik umfassend zu begegnen, wird ein Modell-zentrischer Software Test Lebenszyklus entwickelt, der die üblichen Prozessschritte und Artefakte des klassischen Testens auf der Modellebene umsetzt. Hierzu werden zunächst die konzeptuellen Grundlagen geschaffen, indem die verfügbaren Modellartefakte sämtlicher Domänen in Relation gesetzt werden. Dies bedeutet insbesondere, dass eine Verknüpfung von Modelldaten umgesetzt wird, um eine hinreichende Grundlage für die darauf aufbauenden Prozessschritte des Lebenszyklus zu schaffen. Darüber hinaus wird ein flexibles Metamodell inklusive operationaler Semantik entwickelt, welches unter anderem die Grundlage für die abstrakte Ausführung der Modellartefakte darstellt.

Aufbauend auf dieser Grundlage werden Ansätze für das Testfallmanagement, die automatisierte Testfallgenerierung, die Evaluation von Testfällen und die Qualitätsüberprüfung von Testfällen entwickelt. Im Kontext des Testfallmanagement wird ein Mechanismus realisiert, der die Auswahl, Priorisierung und Reduktion von Testmodellartefakten ermöglicht und diese der Testfallgenerierung bereitstellt. Die entwickelten Konzepte der Testfallgenerierung nutzen diese spezifischen Testmodelle, um mittels etablierter Methoden, welche auf der Modellebene umgesetzt wurden, eine zielgerichtete Testfallmenge zu erzeugen, die gewissen Qualitätskriterien wie z.B. Abdeckungen auf unterschiedlichen Modellierungsdomänen entspricht. Diese Qualitätsanforderungen werden durch den Einsatz einer Mutations-basierten Analyse der ermittelten Testfälle bewerkstelligt, die wiederum auf der Modellbasis aufbaut. Als letzter Bestandteil des entwickelten Modell-zentrischen Software Test Lebenszyklus werden zwei Ansätze vorgestellt, die mittels struktureller Analyse und einer Form der Modellinterpretation eine abstrakte Ausführung der Testfälle im Modellkontext ermöglichen. All die erwähnten Ansätze zur Bewerkstelligung der Problemstellung werden im Verlauf der

Arbeit jeweils in den Kontext verwandter Forschungsarbeiten gesetzt, als auch im Rahmen einer prototypischen Umsetzung im Rahmen des *Architecture And Analysis Frameworks* auf deren Umsetzbarkeit untersucht.

Anschließend werden die beschriebenen Ansätze und deren Konzepte durch qualitative, als auch quantitative Betrachtungen der Teilaspekte evaluiert. Diese Evaluation wird vor dem Hintergrund einiger Anwendungsbeispiele aus der Praxis durchgeführt, wodurch die praktische Anwendbarkeit der Ansatzes gezeigt wird.

Danksagung

Im Kontrast zu den sachlichen Ausführungen dieser Dissertation möchte ich an dieser Stelle ein herzliches Dankeschön und eigene persönliche Worte formulieren. In erster Linie möchte ich meinem Doktorvater Prof. Dr. Bernhard L. Bauer für die Betreuung danken. Durch ihn wurde das Projekt Promotion erst möglich gemacht und auch während der Umsetzung durch wertvolles Feedback bereichert. Insbesondere ist hier das entgegen gebrachte Vertrauen im Sinne des verfügbaren kreativen Spielraums in Kombination mit der Möglichkeit Verantwortung zu übernehmen hervorzuheben, was mich in meiner persönlichen Entwicklung sehr bestärkt hat. Herzlichen Dank Bernhard! Ebenfalls möchte ich mich bei Prof. Dr. Alexander Knapp für das wertvolle Feedback und seine Arbeit als Zweitgutachter bedanken.

Darüber hinaus möchte ich mich bei all meinen Kolleginnen und Kollegen bedanken, die mich in irgendeiner Weise unterstützt haben. Sei es durch angenehme Gespräche in den (Kaffee-)Pausen, notwendige und manchmal erheiternde Diskussionen im Lehre Kontext, teilweise ernüchternde Arbeiten im Projektalltag oder auch die produktiven Nachtschichten im Zusammenhang mit unserem autonomen Fahrzeug, bei denen der soziale Aspekt auch nicht zu kurz kam. In Summe lassen mich all diese Aspekte mit einem weinenden Auge auf die verstrichene Zeit an der Professur zurückblicken, weil mir die Arbeit durchwegs Freude bereitet hat und ich sehr viele Persönlichkeiten zu schätzen gelernt habe. Herzlichen Dank euch allen!

Da meiner Meinung nach eine Promotion nur mit starkem Rückhalt aus dem privaten Umfeld zum Erfolg führt, möchte ich meinen Freunden und meiner Familie herzlich danken. Auch wenn der Kopf nur schwer davon abzubringen war nach Lösungen für Herausforderungen der Dissertation zu suchen, konnte ich im privaten Kontext immer wieder die notwendige Energie schöpfen und zur Ruhe kommen. Insbesondere will ich an dieser Stelle meiner Frau, meinem Sohn und meinen Geschwistern danken, die mir durchwegs den Rücken stärken, im privaten Umfeld “den Laden am Laufen halten” und Tag für Tag mein Herz zum Lachen bringen. Abschließend möchte ich mich bei meinem Vater und meiner leider viel zu früh verstorbenen Mutter, der ich diese Arbeit widme, bedanken, ohne die ich heute nicht dort stehen würde, wo ich stehe, ohne die ich nicht die Persönlichkeit wäre, die ich bin! Herzlichen Dank, ihr seid großartig!

Contents

Part I

MOTIVATION, RESEARCH ITEMS AND OUTLINE

1

Introduction

The invention of the first computer by Konrad Zuse launched a new era. Until then, the performance of extensive calculation tasks had only been possible with human assistance. However, this changed with the introduction and further development of computers in such a way that now developers only implemented the tasks (programs), but the execution was carried out by the machine. Over the years, however, the tasks became more and more demanding, which enormously increased the complexity and volume of the machine code written (see [64]). With the introduction of high-level languages this complexity was counteracted and the level of abstraction for specifying new functionality was increased. The first *Fortran* compiler, in particular, represents a milestone, as it shifted the developers' attention back from technical to algorithmic challenges [26].

The race against Moore's Law, starting in the 1980s with the invention of integrated circuit technologies steadily increased the processing power of computers until today. With the continuous improvement of the available hardware, the possibilities with regard to realizable tasks are continuously increasing. The resulting variety of today's programming languages and their high-level language concepts testify to the constant development which is driven by the increasing complexity of the tasks.

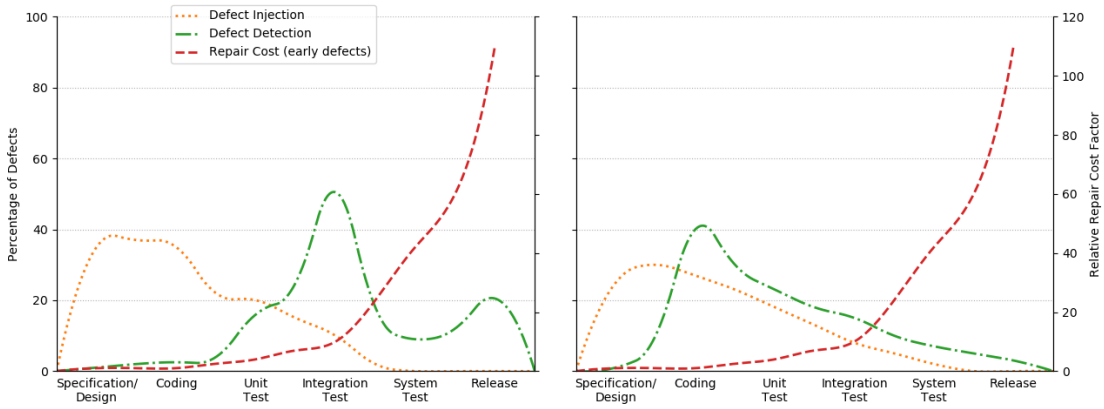
Possibly, we are currently performing another paradigm change. In early phases of development, different modeling variants are used, i.e. to visualize use cases, to sketch a rough design of the software to be developed, or even to specify communication flows between components. No matter which concrete role modeling plays in a development process, it overcomes a certain flavor of emerging complexity. The efforts towards solid solutions, which use modeling as a core technology during software development, are driven by standardization bodies such as the Object Management Group (OMG). The Model-Driven Architecture (MDA) standard explicitly presents an approach on how modeling should be implemented in the construction phases of development [126]. Besides attributes such as improved reusability or better product quality, MDA is often promoted with the automation of error-prone steps in development [69] [160].

In concert with high-level programming languages, the use of Model-Driven Software Development (MDSD) approaches has led to better performance during construction phases of complex systems [28]. However, it turns out, that the construction phases represent only half of the truth and the question of how subsequent or ongoing Verification and Validation (V&V) is applied in such scenarios arises.

1.1 Problem Statement and Research Questions

The state-of-the-art V&V approaches applied in software development are mostly carried out on code artifacts of the System Under Development (SUD) or executable artifacts for the target platform. In an MDSD context, this effectively reduces the set of applicable techniques to either error-prone manual and heavily experience-based approaches, e.g. reviews, or automated formal approaches like model checking, which require substantial knowledge in formalization. Alternatives based on the execution or simulation of early development artifacts show a strong focus on certain specification languages and/or the application context, e.g. Papyrus or Matlab. Most state-of-the-art V&V approaches are based on the code-level, probably leading to unsatisfying feedback loops for corrective tasks.

Especially in traditional development processes, in which few iterations through the development phases are performed, the effects are well studied. Figure 1.1a visualizes Feiler et al.'s summary (see [64]) of multiple studies on software defects and resulting relative costs ([141] [71] [36] [49] [30]).



(a) Standard scenario (based on [64])

(b) Shift Left scenario

Figure 1.1: Correlation of defect injection, detection, and emerging costs

Basically, both diagrams in figure 1.1 are based on three different scales. The left scale shows the *percentage of defects* from 0 to 100, while the bottom scale draws the different development phases, *Specification/Design*, *Coding*, *Unit Test*, *Integration Test*, *System Test*, and *Release*. In addition, the right scale reflects the *relative repair cost factor*, which ranges from 0 to 120.

First, the diagram on the left side encapsulates the information of standard development and test scenario. Thereby, the dotted line shows the distribution of *Defect Injection* through the previously mentioned development phases. Across all the included studies, it turns out, that around 80% of the defects are commonly introduced during *Specification/Design* and *Coding*. Furthermore, around 80% of these defects are only discovered in phases after the *Unit Testing* phase, which is visualized by the dotted and

dashed *Defect Detection* line. Consequently, this has a huge impact on the costs for fixing the defects introduced in the early stages of development, which is represented by the dashed *Repair Cost* curve revealing an exponential growth towards late phases. [64]

In order to reduce the overall costs for fixing defects, a *Shift Left* of V&V tasks, sometimes called *Front Loading*, is desirable. Figure 1.1b visualizes the shift left scenario revealing the main effects. On the one hand, the number of defects detected in late phases is reduced and shifted towards phases like *Specification/Design* and *Coding*, thereby reducing the emerging costs. On the other hand, with the introduction of V&V mechanisms applied in early stages, the relative number of *early defects* is reduced, again having a positive impact on the resulting costs. In addition, the previously exponential growth of the relative *Repair Cost* factor is expected to be slower than in the standard scenario, which is not yet provable. Depending on the underlying development process, the Shift Left is carried out on different levels of granularity, e.g. on a complete development life cycle or during an increment of it. In practice, the Shift Left can be seen by the use of modern approaches like Test-Driven Development (TDD) or the Development and Operations (DevOps) movement.

Complexity Challenge in Software Testing and Insufficient Applicability on Model-Level

Driven by the constantly increasing complexity of software and systems under development, software testing, a concrete methodology for V&V, has become more challenging. Analogous to approaches of MDSD, certain aspects of software testing are already realized by modeling and automating processing steps. In the so-called Model-Based Testing (MBT), steps such as test case creation are performed at the model-level, while the subsequent execution of the test cases against the target system is carried out on the code-level. Especially by deriving the test cases from models, the number of detected defects can be increased significantly, and thus drastically reduces the costs (see figure 1.1a) [144]. Further, MBT can be seen as a partial attempt to perform the shift left of software testing activities towards the early stages of MDSD. However, MBT does not map all artifacts of a Software Testing Life Cycle (STLC) to the model-level, which still limits the early application of testing. Especially in cases where MDA is applied and consequently code is automatically generated from models, there is an additional level of indirection between fault and detectable failure. This leads to the first research question:

Research Question 1
How to carry out a testing life cycle on the model-level to counteract complexity and severe defects, improving the overall quality of development artifacts in MDSD scenarios?

Insufficient Information Separation and Integration Across Different Development Disciplines

The easiest way of applying MBT is to enrich the Software and System Model artifacts with test-specific data. One major drawback of this kind of MBT is the missing divergence in the interpretation of requirements by developers and testers. In scenarios, where code is automatically derived from model artifacts, the test cases derived from the same, enriched model artifact, are not able to detect any functional defects, but defects in the applied code generator. [143] Overall, this reflects a lack of information separation, leading to conceptual problems during testing and unforeseen side effects.

Even if there are separate model artifacts to carry out an advanced MBT approach [143], lots of development artifacts from varying domains like safety or security are available, but not yet taken into account for the improvement of automated testing efforts. The problem of missing integration of information prevails in the area of documentation of complex systems and is often solved by controlled integration [15]. The aspects mentioned in the context of MDSD and MBT results in the next research question.

Research Question 2
What kind of modeling conventions and metamodeling concepts are necessary to effectively apply testing activities in early stages of MDSD?

Test Case Selection, Prioritization and Reduction

In scenarios, where coverage-oriented test suite selection metrics are applied to complex systems, the path explosion problem is even harder and leads to exploding test suites [88]. Especially, in code-based white box testing approaches the test suite is build using selection criteria, subsequently prioritized, and finally reduced by applying sophisticated algorithms, like e.g. genetic algorithms. However, following such a processing approach the complexity is revealed to its full extent. In contrast to algorithmic approaches towards efficient and adaptive test suites, information-driven approaches often manage to deal with complexity at its origin, which raises the following research question:

Research Question 3
How can an intuitive and holistic data-driven test case management approach (selection, prioritization, and reduction) on the model-level lead to manageable test suites reflecting the tester's mindset?

Limited Adaptability of Straightforward Test Generation Criteria

In a white box testing context, control flow or data flow-oriented test suite selection criteria like statement coverage, are a prominent way to control test case generation. This also applies to the MBT context, regardless of the concrete manifestation of the involved model artifacts. Metrics are usually applied to the entire model since this is either prescribed by regulatory restrictions (e.g. by ISO26262 or DO178B/C), or due to missing insights into the instances under consideration. Indeed, this represents a quite pessimistic and bloated approach if the resulting size of the test suite is hardly manageable. Especially, in cases where the test engineer has sufficient insight and context knowledge of the use case, the following research question needs more investigation:

Research Question 4
How can better adaptability and context sensitivity of test generation based on coverage criteria and measures on the model-level reduce the overall complexity?

Limited Test Execution Mechanisms in Early Stages of Development

An essential concept of software testing is monitoring the behavior of an executable artifact of the System Under Test (SUT). As we have seen in the introductory part of this chapter, defects revealed in the late stages of development do have a significantly higher impact on the overall costs of the project, than defects detected early. Besides, we pointed out the high number of concept-level defects, e.g. requirements defects, compared to other development phases. To counteract these issues, an execution mechanism for the model-level is desirable to support another shift of software testing activities towards models. However, the palette of model execution and simulation frameworks is mostly bound to specific modeling approaches and further not designed to deal with model artifacts development, i.e. incomplete or faulty artifacts. Therefore, we identified the following research question:

Research Question 5
How can an automated mechanism for the verification of test cases based on (probably incomplete) models of the SUD reduce the emerging test complexity?

Misunderstanding of Test Case Quality

As previously mentioned, integrity in a sense of coverage often represents a major quality attribute of a test suite, e.g. in safety-related standards. However, it is questionable whether a test suite with sufficient coverage is effective in terms of test case quality. Therefore, many studies investigated a correlation between coverage and effectiveness of a test suite [98][87]. It turned out, that none of the studies could provide evidence

for that. Consequently, we see a need for another technique, which can determine the quality of test cases using their defect detection capabilities. On the source code-level, as well as on dedicated models, there are mutation testing frameworks, sufficient for this job. In a more general modeling context the following research question needs to be answered:

Research Question 6
How can mutation-based testing concepts together with a mechanism for test case execution applied on the model-level improve quality of automatically generated test suites?

1.2 Concepts and Objectives

In the previous section, we addressed several issues which interfere with a targeted application of V&V approaches in an MDSD context, particularly testing. Continuing the research questions presented in the previous chapter, the remainder of this chapter outlines possible approaches to solutions, from which the concrete objectives are derived. In particular, the research questions and objectives with identical numbers constitute a unit, e.g. Research Question 3 corresponds to Objective 3.

However, before the concrete ideas for solutions and the resulting objectives are discussed, the overarching challenges are first addressed:

1. *High complexity in software testing due to increased complexity of developed software/systems*
2. *Insufficient semi-formal and automated testing approaches to resolve costly defects at their origin*

To make the complexity of software tangible, a large number of metrics have been developed to identify potential weaknesses and initiate appropriate countermeasures [158]. Moreover, metrics for specific model artifacts were presented, applicable in MDSD [51]. Besides the numeric representation of complexity through such metrics, the meta aspects *difficulty of understanding the software/system* and *difficulty of identifying and correcting defects* in our opinion represent the most basic indicators for test complexity [94]. We believe, that a more result-oriented way of MDSD and testing represents an effective countermeasure to manage complexity. Therefore, based on the idea of Design For Testability (DFT), which originated in the context of hardware design, a strict alignment of the applied development paradigms across development disciplines is desirable, which requires a continuous facing of complexity by appropriate countermeasures.

Kopetz defines the following strategies to counteract complexity [114]:

1. **Abstraction** (also referred to as *conceptual chunking*):
   In most cases, a higher-order concept is used, which omits information that is

irrelevant in the current problem context which can be achieved by Domain-Specific Languages (DSLs) or modeling in general.
2. **Partitioning** (also referred to as the *separation of concerns*):
   Makes use of spatial division of the problem scenario, to study independent parts in isolation. Tightly coupled to the well-known divide-and-conquer strategy has its limits when emergent effects take place.
3. **Segmentation** (also referred to as *temporal decomposition*):
   Divides complex behavior along the temporal dimension, splitting up complex tasks into a sequence of partial tasks.

The application of such strategies throughout the model-centric interpretations of the traditional STLC, we expect to result in an improved overall quality of the final product, and reduced costs.

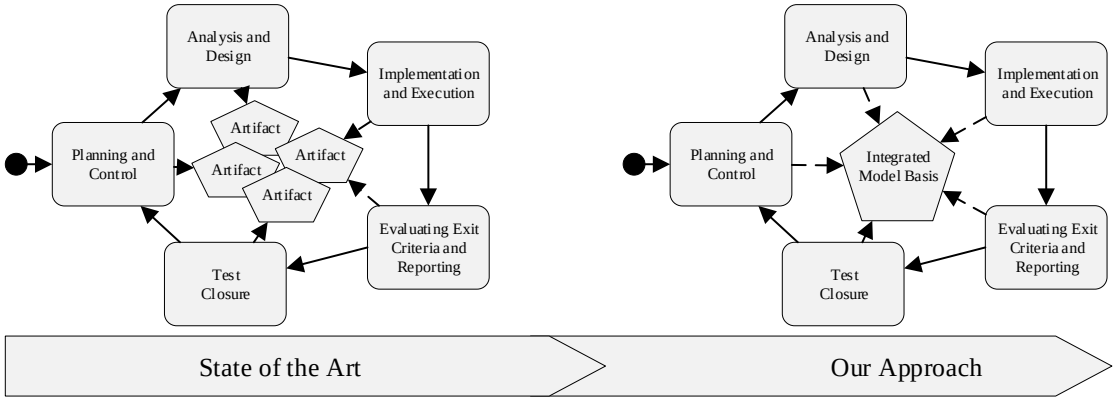


Figure 1.2: Illustration of general approach (see objective 1)

In order to achieve this goal, we attempt to map and interpret all sub-steps of the well-known STLC to the model-level. Figure 1.2 shows the intended switch from a process step-oriented way of information generation and serialization (also from a development perspective) towards a model-centric approach, where all the process steps to profit from an integrated model basis. The way in which process steps are implemented at the model-level depends strongly on the applicability or necessity of certain activities in this context. In particular, the existing approaches from the context of MBT need to be reasonably continued. In addition to these concepts, the views of various standardization authorities in the field of application should also be taken into account and incorporated into the implementation.

A noticeable improvement in the overall quality of the product created, as well as a significant reduction in cost-intensive defects, is expected from this integrated approach. This is achieved by shorter feedback loops and certain agility in MDSD. This ambitious goal is phrased in the following objective.

Objective 1
Manage complexity and improve quality through a Model-Centric Software Testing Life Cycle (MCSTLC) with sufficient tool support.

In the following, concrete components of the planned solution path are examined more closely, their core aspects are pointed out, and concrete objectives are derived. First, we focus on the integrated model basis, representing the foundation for all further steps.

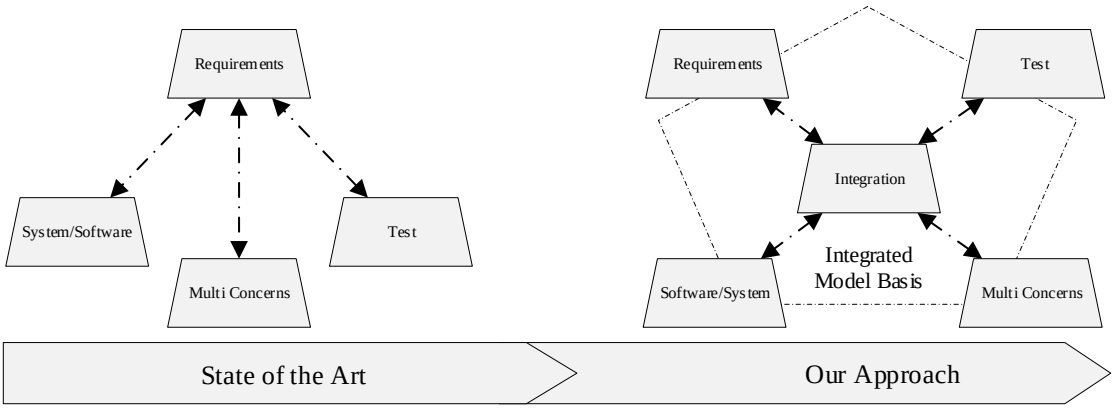


Figure 1.3: Abstract illustration of the general approach from traditional structures towards an Integrated Model Basis

In addition to the application of the abstraction concept, we emphasize the advantages of partitioning the problem. I.e. a clean separation of concerns with regard to the created development artifacts, e.g. already available models divided along with the development domains (see figure 1.3). Moreover, domain experts can use their specific tooling or domain-specific (modeling) languages as usual. No multi-purpose view concept is applied, but specific metamodels are used for each development domain. However, this does not exclude the application of view mechanisms within a certain development domain. Information for the interaction across different development domains as well as information resulting from different processing steps (metadata, analysis results, results of previous iterations to regression information) is explicitly stored in an Integration Model. This represents the flexible link between model artifacts including requirements (see the left side of figure 1.3). Further, information can be used to improve the redesigned testing activities in the context of the MCSTLC. Thereby, a more targeted variant of testing is expected, leading to objective 2:

Objective 2
Reduce complexity by model-level separation of concerns and expand knowledge by controlled information integration.

1. *Apply the separation of concerns to involved disciplines, e.g. for development and test.*

2. *Roll out a controlled integration of model artifacts concerning MCSTLC activities.*
3. *Develop concepts for flexible application of MCSTLC activities, independent of the underlying development setup.*

As soon as the integrated model basis is available in an initial version, the execution of the MCSTLC starts. In a conventional STLC, planning is usually based on the requirements, which is the basis for the creation of test cases (see the left side of figure 1.4). This

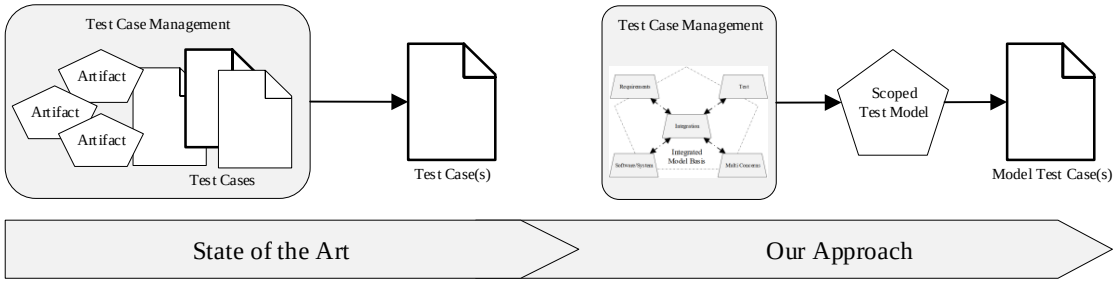


Figure 1.4: Illustration of the shift towards the application of Test Case Management approaches on the model-level

activity is referred to as *Test Case Management* and includes the selection, prioritization, and reduction of a test case set. Moreover, this ensures that the resulting set of test cases reflects the test focus regarding the current development phase and requirements. Our model-centric approach uses abstraction, partitioning, and segmentation, to rearrange the STLC on the way to a test case set. Based on the integrated model basis, including an isolated Test Model specifying the intended behavior of the overall system, a strongly focused extract is derived. Namely, a *Scoped Test Model* serves all the purposes of the legacy version and reflects the tester's mindset (see right side figure 1.4). Subsequent steps transfer the model excerpt to a concrete set of test cases, resulting in objective 3:

Objective 3
Develop a test case management approach based on an integrated model basis instead of requirements and planning artifacts.

1. *Specify a DSL to reflect the tester's mindset for model-level test focus specification.*
2. *Establish a mechanism for scoping Test Models based on the test focus.*

As we mentioned above, the *Scoped Test Model* artifacts represent the starting point for the generation of a concrete set of test cases. Core concepts of traditional MBT are optimized about the strongly focused source artifacts and through a context-sensitive way of Test Case Generation (TCG). Again the abstraction concept is applied in concert with a partitioning of the problem domain. Specifically, the partitioning of the extracted model artifact is aligned to the modeled integration levels, resulting in a set of Scoped Test Models per integration level (see figure 1.5).

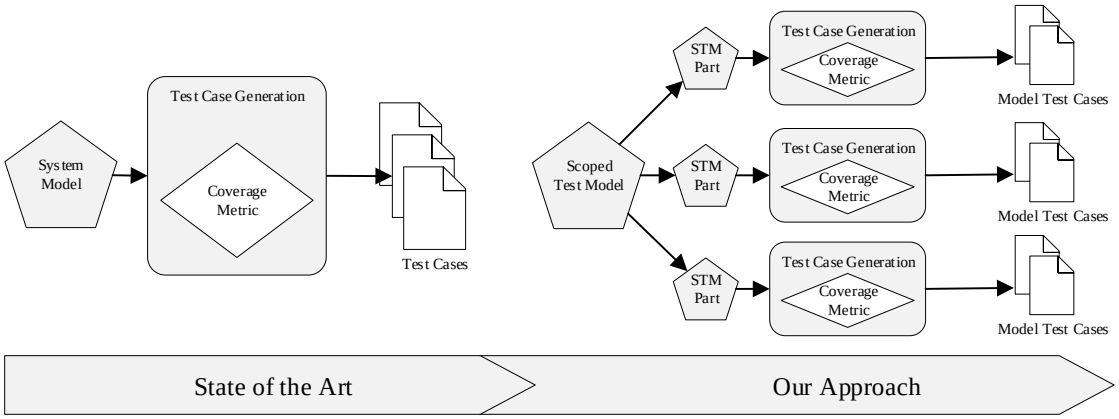


Figure 1.5: Illustration of the shift towards a model-level Test Case Generation approach

Therefore, the algorithm for test case generation can treat the individual models context-sensitive and consequently apply appropriate metrics. The applied criteria refer to traditional control flow and data flow-aware properties of the source model. The decision about the most suitable metric is further guided by information from linked domain-specific models of the integrated model basis. In addition, regression and quality information from previous iterations of the MCSTLC may be taken into account. Below, an objective is defined reflecting the mentioned aspects.

Objective 4
Reduce the test complexity by selective adaption of test quality and test integrity aware test generation criteria.

1. *Lift the code-based generation criteria to the model-level.*
2. *Implement a mechanism for continuous adaptation of test generation criteria based on integrated model information and feedback.*

Having generated the set of test cases reflecting the current test focus, the most essential step for software testing takes place, namely the test case execution. To reside on a unified level of abstraction, the execution mechanism is lifted to the model-level (see figure 1.6).

At this point, a model representation independent of the input metamodel is desirable to achieve broad applicability. Supporting such an execution mechanism, model transformations are used, having the ability to streamline potentially vague execution semantics. The Abstract Test Execution (ATE) itself is carried out by a structural analysis approach and a hybrid interpreter, both operating on the analysis-specific internal model representation. In early phases of development, the control flow-aware approach is applied to handle intermediate and therefore potentially incomplete model artifacts. In later phases, where data flow information is partially available, the hybrid

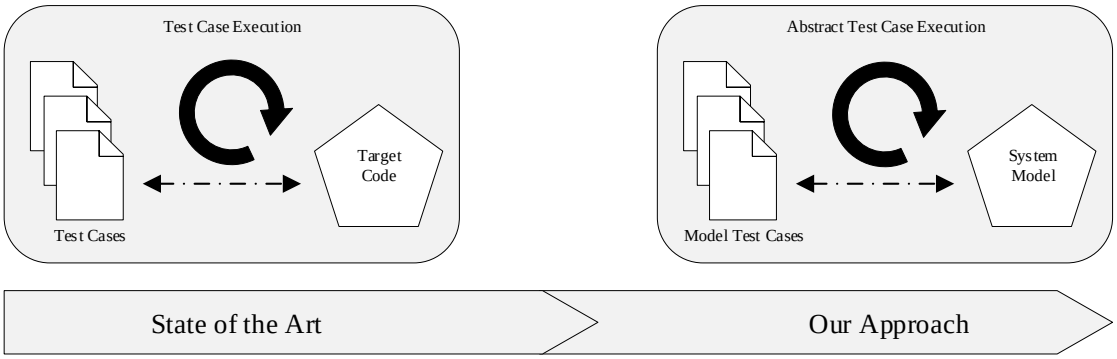


Figure 1.6: Illustration of the intended shift towards an Abstract Test Execution approach

model interpreter approach performs a more sophisticated version of ATE. Both approaches share the goal to serve as much information about the ATE as possible and finally derive a test verdict for each test case. The major points are represented by the following objective:

Objective 5
Manage test complexity in early stages by model-level execution of abstract test cases.

1. *Develop approaches for abstract verification of test cases based on an internal model representation.*
2. *Develop operational semantics for the internal processing model.*
3. *Align the test and System Model information based on the internal processing model.*
4. *Develop a mechanism for abstract verification of model test case against the system model.*

In addition to the adaption and modification of traditional STLC activities to the model-level, the MCSTLC is extended by a vital concept for improving the adequacy of a test suite. A common inaccuracy in the testing context is putting the term *adequacy* on a level with *quality*. However, the traditional STLC does not foresee an activity, which automatically performs quality assurance and control for the test cases themselves. This is usually done manually using reviews (see figure 1.7) or by simplified metrics, such as the integrity of a test suite, which does not necessarily result in good test cases and therefore is a misconception.

Mutation Analysis can fix this shortcoming by performing automated quality checks. Adopting this approach, the concept of abstraction can be applied, and the code-based concepts are lifted to the model-level. Major improvements of our model-level variant are related to the discovery of an appropriate set of mutation operators still reflecting common faults, consequently generating an effective set of mutants, and finally making use of the integrated model basis to tune scalability. These aspects are reflected by the following objective:

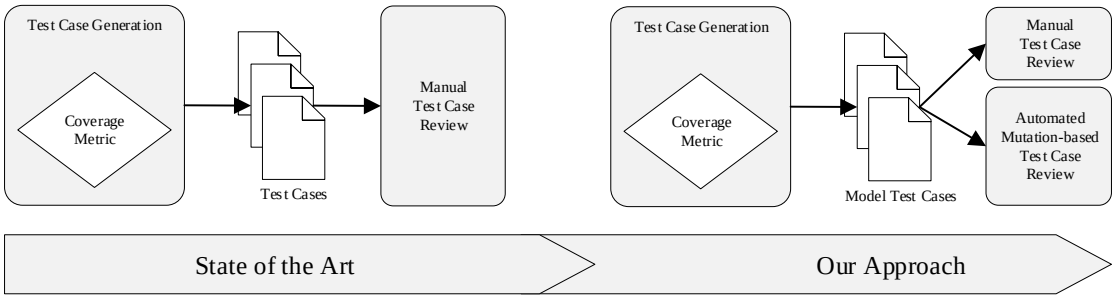


Figure 1.7: Illustration of an automated quality assurance for generated model test cases

Objective 6
Increase the test case quality by a model-level mutation-based quality rating of test cases.

1. *Determine the set of model-level mutation operators.*
2. *Determine rules for targeted mutant generation based on an integrated model basis.*
3. *Lift the code-based mutation analysis approach to the model-level and improve scalability.*

2

Research Items

Within this section, we present the set of research items contributing to the work described in this thesis. Hereby, we group the set of research artifacts by their scientific context and relevance in the main chapters. The first section (section 2.1) presents the set of peer-reviewed conference papers and subsequent journal contributions, building the core of the thesis. Furthermore, section 2.2 elaborates on related research projects we participated in during the work on this thesis. The last section (section 2.3) includes a selection of supervised bachelor and master theses affiliated to the tackled problem domains.

2.1 Publications

During this section, we elaborate on the peer-reviewed publications in the context of this thesis. Therefore, we first present a compact version of the publication's contents. Furthermore, the personal contribution to the research item is pointed out in more detail. Finally, we give the reader some advice on chapters of the thesis linked to the respective publication.

A Domain-aware Framework for Integrated Model-based System Analysis and Design
Conference Paper [151]

Abstract: The increasing complexity of modern embedded systems demands advanced design and development methods. Incremental evolution of model-based engineering practice has led to heterogeneous tool environments without proper integration and exchange of design artifacts. These problems are especially prevalent in tightly regulated domains, where an independent assessment is required for newly developed products, e.g. in automotive or aviation systems. To address these shortcomings of current engineering practice, we propose a holistic model-based approach for the seamless design and development of an integrated system model. We describe an embedding of a variety of domain-specific modeling languages into a common general-purpose modeling language, in order to facilitate the integration between heterogeneous design artifacts. Based on this conceptual modeling approach, we introduce a framework for automated

model-based analysis of integrated system models. A case study demonstrates the suitability of this modeling and analysis approach for the design of a safety-critical embedded system, a hypothetical gas heating burner.

Personal Contribution: In general, the publication represents the joint work of the authors Rumpold and Proell. Starting with section 2 *A Domain-Aware Approach For System Modeling*, the conceptual work on the model to model transformations and orchestration of domain-specific modeling languages represents my contribution. Further, section 3 *Model-Based Architecture And Analysis Framework* reflects the technical realization of the presented concepts. Apart from the base functionality, which was developed together, the project-specific functionality for orchestration and test-related topics (ReTeC, T3) represents my contribution. Finally, section 4 *Case Study: Reliability Model For A Gas Heating System* was jointly developed by the two authors, including the creation of the case study as well as its discussion.

Related Parts in Thesis: Section 7.1.3, Section 7.3

Applying Integrated Domain-Specific Modeling for Multi-concerns Development of Complex Systems
Journal Paper [148]
Extended Version of [151]

Abstract: Current systems engineering efforts are increasingly driven by trade-offs and limitations imposed by multiple factors: Growing product complexity as well as stricter regulatory requirements in domains such as automotive or aviation necessitate advanced design and development methods. At the core of these influencing factors lies a consideration of competing non-functional concerns, such as safety and reliability, performance, and the fulfillment of quality requirements. In an attempt to cope with these aspects, incremental evolution of model-based engineering practice has produced heterogeneous tool environments without proper integration and exchange of design artifacts. In order to overcome these shortcomings of current engineering practice, we propose a holistic, model-based architecture and analysis framework for seamless design, analysis, and evolution of integrated system models. We describe how heterogeneous domain-specific modeling languages can be embedded into a common general-purpose model in order to facilitate the integration between previously disjoint design artifacts. A case study demonstrates the suitability of this methodology for the design of a safety-critical embedded system, a hypothetical gas heating, with respect to reliability engineering and further quality assurance activities.

Personal Contribution: In general, the publication represents the joint work of the authors Rumpold and Pröll. Due to the fact, that this publication represents an extended version of the previously introduced conference paper, the structure and contents are almost the same. Sections 2-4 (*A Domain-Aware Modeling Approach for Embedded System Engineering*, *A Model-Based Architecture and Analysis Framework* and *Case Study: Design and Evaluation of a Gas Heating System*) follow the distribution of topics given by the con-

ference paper, which holds for the work contributed by the two authors. Contributions added during the rework were created by both authors.

Related Parts in Thesis: Section 7.1.3, Section 7.2, Section 7.3

Toward A Consistent And Strictly Model-Based Interpretation Of The ISO/IEC/IEEE 29119 For Early Testing Activities

Special Session Paper [146]

Abstract: Effective and sufficient testing has always been a challenging task in software development. The ongoing increase of software complexity forces developers and testers to make extensive use of the concept of abstraction, thereby leading to model-based approaches. Further, standardization organizations aim for harmonized process templates to assure a certain quality level of the processes behind. In order to combine the process-level advice as well as the concept-level advice, we aim for a consistent and strict application of model-based methodologies throughout the testing processes, introduced by the ISO/IEC/IEEE 29119 standard for software testing. After a brief introduction to the standards content and a critical view on it, we focus on our model-based interpretation of the postulated processes. Thereby, we extend the original idea of model-based testing, incorporating the separation of concerns on the model-level, to form a broad information basis. Subsequent activities are aligned with these concepts, in order to make sure a purely model-based testing life cycle, with respect to consistency and quality of development artifacts. Following the related work of impacted research areas, we end up with a conclusive statement on the intended combination of approaches.

Personal Contribution: This publication exclusively contains research of the first author, who also is the author of this thesis.

Related Parts in Thesis: Section 1.2, Chapter 8 - chapter 11 in a general context

A Model-based Test Case Management Approach For Integrated Sets Of Domain-Specific Models

Workshop Paper [145]

Abstract: Due to rapid improvements in the area of embedded processing hardware, the complexity of developed systems constantly increases. In order to ensure a high quality level of such systems, related quality assurance concepts have to evolve. The introduction of Model-Based Testing (MBT) approaches has shown promising results by automating and abstracting multiple activities of the software testing life cycle. Nevertheless, there is a strong need for approaches supporting scoped test models, i.e. subsets of test cases, reflecting specific test purposes driven by risk-oriented development strategies. Therefore, we developed an integrated and model-based approach supporting test case management, which incorporates the beneficial aspects of abstract devel-

opment methodologies with predominant research for test case management in non-model-based scenarios. Based on a new model artifact, the integration model, tasks like cross-domain information mapping and the integration of domain-specific KPIs derived by analyses favor the subsequently applied constraint-based mechanism for test case management. Further, a prototypical implementation of these concepts within the Architecture And Analysis Framework (A3F) is elaborated and further evaluated based on representative application scenarios. A comparative view on related work leads to a conclusive statement regarding our future work.

Personal Contribution: This publication exclusively contains research of the first author, who also is the author of this thesis.

Related Parts in Thesis: Chapter 8

Towards Abstract Test Execution in Early Stages of Model-Driven Software Development
Conference Paper [83]

Abstract: Over the last decades, systems immanent complexity has significantly increased. In order to cope with the emerging challenges during the development of such systems, modeling approaches become an indispensable part. While many process steps are applicable to the model-level, there are no sufficient realizations for test execution yet. As a result, we present a semi-formal approach enabling developers to perform abstract test execution straight on the modeled artifacts to support the overarching objective of a shift left of verification and validation tasks. Our concept challenges an abstract test case (derived from test model) against a system model utilizing an integrated set of domain-specific models, i.e. the omni model. Driven by an optimistic data flow analysis based on a combined view of an abstract test case and its triggered system behavior, possible test verdicts are assigned. Based on a prototypical implementation of the concept, the proof of concept is demonstrated and further on put in the context of related research.

Personal Contribution: The first author was a master student of the second author and worked on the defined task, which was previously solved on a conceptual level by the second author. This publication exclusively contains research of the second author, who also is the author of this thesis. All in all, the research presented in this paper remains the intellectual property of the second author.

Related Parts in Thesis: Chapter 10, Section 10.2.2

Abstract Test Execution for Early Testing Activities in Model-Driven Scenarios
Journal Paper [147]
Extended Version of [83]

Abstract: The constant improvement in the performance of computing units enables them to handle increasingly complex tasks. This usually demands for more complex software, the development of which is difficult to achieve using traditional approaches. With concepts like model-driven software development this problem can be defused for the constructive phases. However, new challenges arise for the testing of development artifacts. To be able to perform a real shift left of verification and validation tasks in this context, we present a semi-formal approach that enables us to execute test cases against the system under development on the model-level. Grounded on an Integrated Model Basis which is created and maintained during development, test reports are automatically derived. This opens up a wide range of possibilities for early and targeted troubleshooting. After a discussion of the algorithmic implementation, the presented approach is categorized.

Personal Contribution: Since this is an extended version of the publication mentioned above, the contents are largely identical. In addition to a much more formal presentation, a different running example was used. Furthermore, the mutation-based evaluation was replaced by a critical discussion and qualitative evaluation. Regarding the assignment of individual parts to the participating authors, the same applies to the original publication. The content dealing with details of the technical implementation is attributed to the second author. All other content, especially conceptual content, is to be assigned to the first author.

Related Parts in Thesis: Chapter 10, Section 10.2.2

2.2 Research Projects

Throughout this section, the projects in the context of this thesis we were involved in, are presented. Here, we first present a compact view on the project's topics revealing the cross-cutting concerns with this thesis. Further, we point out the research of our working group, especially the author's personal contribution to the project results. Finally, we give the reader some advice on chapters of the thesis affiliated with the respective project results.

Reduction of Test Complexity (ReTeC) [166]

Summary: Within the context of the ReTeC project, a holistic, model-based approach for the development and testing of embedded systems was developed. Starting from a set of requirements, the model-based development of the application parts (structure,

data, functionality) as well as the model-based test-specific artifacts, are to be stored in a similar, domain-spanning model, the so-called Omni Model. Through this integral storage of information around the system to be developed, statements about quality can be made at an early stage through different variants of the "in-the-loop" simulation. To evaluate the underlying concepts about their suitability, a prototype implementation was realized. The subsequent integration into a toolchain represented the conclusion of the desired work in the context of the project.

Personal Contribution: One of the main contributions to the ReTeC project is given by the development of a conceptual basis for the integration of heterogeneous model-based development artifacts. Therefore, I worked on an Omni Model approach, which is about linking and maping model information across domains. The central model artifact is the Integration Model, which specifies additional information to integrate the model data. Based on the integrated set of model artifacts a methodology aiming for the reduction of test complexity has been developed. The complexity reduction is achieved by an intelligent scoping mechanism analyzing the integrated model data while taking the tester's focus into account. To give a proof of concept, the conceptual work was implemented in a prototypical way, further improved toward the first version of our Architecture And Analysis Framework (A3F).

Related Parts in Thesis: Section 7.3, Chapter 8, Chapter 9

Modellgetriebene Software Entwicklung für Funktionale Sicherheit von Automatisierungslösungen (MDSD4SIL) [165]

Summary: Within the cooperation project MDSD4SIL a methodology and toolchain for the model-driven development of safety-critical embedded systems according to IEC 61508 was developed. The core of the development approach is the continuous modeling of the non-functional quality aspects of a system in a common System Model. This consistency enables the analysis of the system to be developed with regard to domain-specific requirements such as functional safety, timing, reliability, and information security. The demonstrator developed in the course of the project allows the modeling of such systems in the commercial modeling tool Enterprise Architect in connection with the radCASE solution of the project partner IMACS. An independent analysis framework provides the necessary infrastructure for performing model analyses on the system via a web service. A final evaluation demonstrated the suitability of the developed solution for use in the modeling of safety-critical systems on the basis of a typical application case.

Personal Contribution: In addition to the conceptual work in the context of the ReTeC project, I extended the Omni Model approach. This involved the extension and generalization of the underlying metamodel concepts towards a flexible set of participating development domains. In this case, models for safety-related concerns, e.g. fault trees were the driving force during the partial redesign of the original concepts. Consequently, I updated the prototypical implementation and synchronized it with my col-

leagues, which lead to the first stable version of the Architecture And Analysis Framework (A3F).

Related Parts in Thesis: Section 7.3, Chapter 7

Test the Test (T3) [167]

Summary: The aim of the project was to enable an automated evaluation of tests with regard to their quality ("Test the Test", T3) by means of fault injection as well as by mutations of the System under Test (SUT) both on the software side and on the hardware side. These evaluations serve to improve the quality of the tests. I.e. code is deliberately and systematically affected by errors in order to evaluate corresponding tests. For this purpose, existing approaches for software and hardware tests were supplemented by a quality analysis and semi-automatic quality improvement of the test cases in order to meet the ever-increasing quality requirements of embedded systems. In addition to the classical approaches for determining test quality, T3 aimed at efficient possibilities for the evaluation of tests. On the one hand, this should be done by evaluating the tests already in the early phases (modeling) of development (design time). On the other hand, a (semi-)automatic improvement of the test quality is to be achieved by suitable adaptation and a combination of classical code metrics. These mechanisms were conceived in a similar way across different levels of abstraction. The developed concepts were prototypically realized within the project and integrated into the existing software and hardware test tools of the project partners. The evaluation was carried out on the basis of some case studies from the addressed areas.

Personal Contribution: In general, this project followed a quite different overall structure and work distribution leading to fewer interaction points with the predominant industrial toolchain. Nevertheless, carrying on the developed concepts and prototype from the research projects MDSD4SiL and ReTeC, I developed another metamodel for internal analysis purposes. Based thereon, a concept for the flexible mutation of System Models, internally using the new analysis metamodel, was designed. Further, I worked on a mechanism for the abstract execution of test cases against the System Model, in order to gain early indications for included bugs. The orchestration of all the developed functionality across projects lead to a mechanism enabling applicants to continuously improve test suite quality. Finally, I managed to complete the implementation of all the concepts within the A3F framework, which helped to evaluate the developments alongside industrial as well as research case studies.

Related Parts in Thesis: Section 7.3, Chapter 11, Chapter 10

2.3 Supervised Thesis

In this section, we present a set of supervised work in the context of this thesis, which means that the topics, as well as the major conceptual decisions, were developed by the

author of this doctoral thesis. In order to point out the role in the context of my thesis, each of the supervised work items is introduced by a brief summarizing paragraph. Finally, we directly address the related sections of the thesis.

Funktionale Absicherung auf dem Weg zum autonomen Fahren am Beispiel eines Autobahnpiloten [59]

Summary: The master thesis elaborated a holistic concept for functional safety analyses in the context of automotive driver assistance systems. Again, the foundations in the areas of automotive development methodologies, safety-related analysis methodologies, as well as hardware-specific safety concerns were presented. The conceptual work on the structured analysis of a automotive assistance system on the basis of a functional as well as a technical safety concept revealed potential for future research in the area of integrated analysis of so far disjoint domains. Finally, a recent case study from the automotive industrial context gave evidence for the adequacy of the work done.

Related Parts in Thesis: Chapter 7 show case for a model artifact which may participate in the Omni Model (fault modeling)

Abstract Execution of Graph-Based Test Descriptions in Model-Driven Software Development [118]

Summary: Due to the economical need to detect errors in software development processes as quickly as possible, the master thesis worked on testing in early stages of model driven software development. In this thesis, a novel approach for abstract execution of test descriptions was presented and prototypically implemented in the architecture and analysis framework. Instead of processing contained instructions, the abstract structure of model artifacts is brought to execution. By comparing model graphs of system and test, structural differences can be identified that indicate discrepancies between actual behavior and intended behavior of the System Under Test (SUT). These indicators can be processed and classified to give the tester feedback for early test set assessment and an indication for potential problems.

Related Parts in Thesis: chapter 10, section 10.2.1

Datenfluss-basierte abstrakte Testausführung in der Modell-basierten Softwareentwicklung [131]

Summary: Continuing the work on the problem domain of the work about "Abstract Execution of Graph-Based Test Descriptions in Model-Driven Software Development", this master thesis elaborates an approach taking into account the data flow of system

and test models. Based on integrated sets of system and test model artifacts a methodology for the systematic evaluation of abstract test case specifications has been developed. A combined view on the graph-based specifications of data flow allows us to draw conclusions about the discrepancies between the modeled system and its respective test model. Along an industrial case study from the automotive domain, a proof of concept as well as the potential for future applications has been illustrated.

Related Parts in Thesis: Chapter 10, Section 10.2.2

Model-to-Model Transformationen im Kontext modell-basierter Software- und System-Analysen [101]

Summary: In this thesis, model transformations in the context of model-based software and system analyses were investigated. The focus was on the Systems Modeling Language (SYSML) which was integrated into the A3F within the scope of the work. In particular, model transformations for the internal analysis-specific metamodels were realized. Starting from a modeling tool specific input representation, the first transformation towards an internal and simplified version of the SYSML metamodel were conducted. In a second step, the simplified SYSML model was transformed to the internal analysis-specific version, which is used to elaborate structure as well as data related aspects of the system model. The developed functionality was examined alongside a running example, namely the Ceiling Speed Monitor (CSM), from a former research group. A final discussion about the newly integrated concepts closes the conducted work.

Related Parts in Thesis: Chapter 7, Section 7.3, Chapter 10

Model-to-Model Transformationen zur Adaption Modellzentrischer Testmechanismen am Beispiel U2TP [102]

Summary: Model-based testing provides an approach to extend the test process of a system with models. In this context, the Object Management Group (OMG) defines the modeling language UML Testing Profile 2 (UTP2) which is used for describing and visualizing test artifacts. This thesis gives an overview of model-based testing, metamodeling and model transformations with a special focus on the OMG standards. After a short introduction of the Eclipse Modeling Framework and the UTP2, three metamodels are presented. Based on these metamodels, two model-to-model transformations are proposed, which convert a UTP2-conform visual model to a model used for further analysis. The thesis concludes with possible concepts to extend the metamodels and model transformations.

Related Parts in Thesis: Chapter 7, Section 7.3, Chapter 10

3

Outline

In this section, an overview of the contents of this thesis is given. The general structure and coherence of the parent chapters can be seen in figure 3.1.

In the first part of the thesis (part I) first, a motivation for the topic is given, the problem as well as the research questions/objectives are presented, and finally relevant research projects and works are discussed.

In the second part of the thesis (part II), the necessary foundations for the concepts presented in the main part are explained. First, general concepts and terms from the modeling environment are discussed (chapter 4). Furthermore, some central approaches of verification and validation of systems are shown (chapter 5), whereby the focus is on testing (section 5.1) considering the interaction with the modeling world (section 5.2).

The third part represents the main part of the thesis, with the components and fundamentals of the MCSTLC (part III). Thus, an overview of the developed concept and its components is given (chapter 6), before a running example is introduced, used throughout the thesis to illustrate the concepts (section 6.2). Thus, as the first substantive part of the MCSTLC, the model basis and the concept of linking different modeling domains are discussed (chapter 7). On the one hand, possible modeling languages, a representative selection of modeling domains, and the metamodel used for information integration are presented (section 7.1). On the other hand, the internal model representation developed specifically for processing the available model information and the concepts for model transformation are explained (section 7.2). As a counterpart to the presented theoretical foundations of the MCSTLC, the prototypical realization is discussed in section 7.3. The final section presents related work, a summary, and an outlook (section 7.4 and section 7.5).

Building on the investigated fundamentals, the remaining chapters of the main part deal with the individual process steps and the concepts developed for this purpose. In chapter 8, the concept for Test Case Management is presented at the model-level, starting with a classification in the overall process including necessary preconditions. Afterward, in section 8.2, the algorithmic implementation is explained before the prototypical realization is discussed (section 8.3). In section 8.4 research work in the context of the approach is elaborated before a summary including an outlook is given (section 8.5).

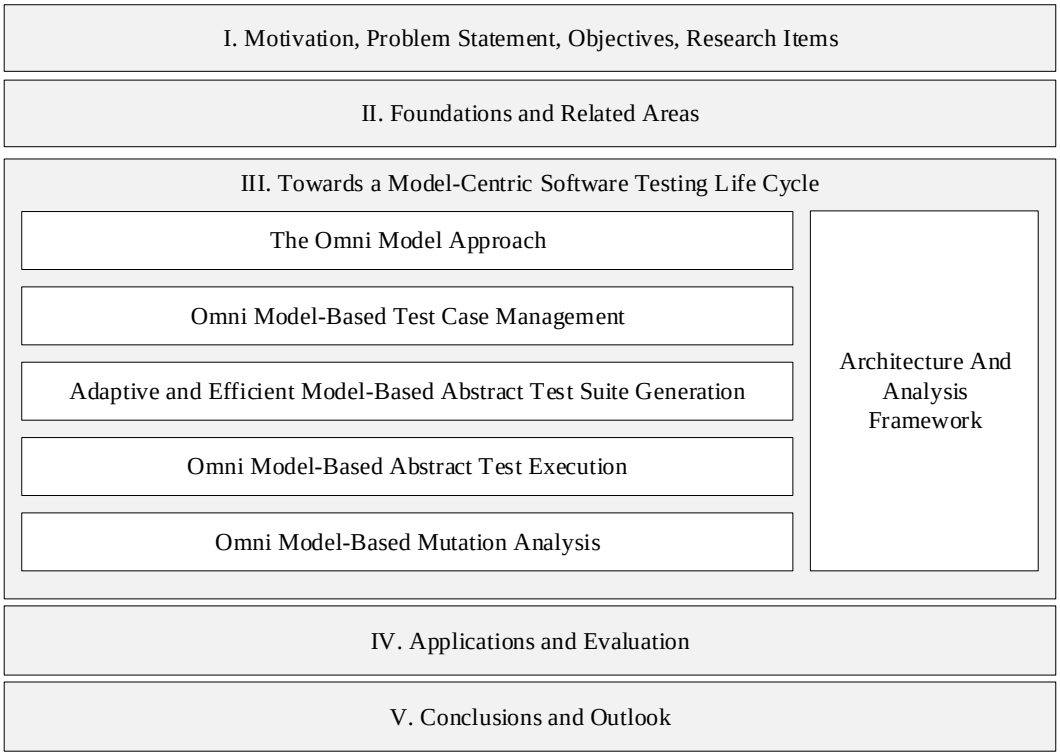


Figure 3.1: Overall structure of the thesis

Chapter 9 has an analogous structure, where the Test Case Generation based on the presented Omni Model is considered. As an introduction, the context and necessary preconditions of the approach are discussed (section 9.1). After the comprehensive introduction of the concept in section 9.2, section 9.3 deals with the implementation in the context of the prototypical Architecture And Analysis Framework. This section concludes with a review of related approaches in section 9.4, as well as a summary and outlook for further topics in this area (section 9.5).

In the context of chapter 10, our concept for Abstract Test Execution is presented. After the introductory part has dealt with the boundary conditions (section 10.1), the presentation of the concepts defines two different approaches to the problem. On the one hand, in section 10.2.1 a concept is presented that works on structural information and is thus suitable for the early stages of development. On the other hand, section 10.2.2 presents a data flow-based concept that processes more detailed information in addition to structural information and is thus designed for advanced phases of model-based development. Further on, the implementation of the two concepts in A3F is discussed (section 10.3), before concluding again with alternative approaches (section 10.4).

The last chapter of the main part represents the concepts for Mutation Analysis (chapter 11). First, as in all sections of the main part, the embedding of the process step in the context of MCSTLC is presented and the interfaces are discussed (section 11.1). In the course of section 11.2, the implementation of Mutation Analysis in the given modeling

environment is discussed, which again ends with the realization of the demonstrated functionality in the context of A3F (section 11.3). Before a concluding discussion of the concepts including an outlook on further topics is given, section 11.4 highlights similar research projects.

The following part IV comprehensively presents and evaluates some application scenarios of the MCSTLC based on an Omni Model. First, the case studies considered are discussed and how they are realized in the context of the Omni Model (chapter 12). In the following chapters, the individual process steps are first evaluated separately with respect to the presented functional scope (section 13.1 to section 13.4, before the entire MCSTLC is evaluated again in chapter 14.

In part V, a summary of the presented components of the approach is drawn, which is supplemented by future application scenarios in the overall context.

Part II

FOUNDATIONS AND RELATED AREAS

4

Model-Driven Software Development

MDSD represents a style of software development, which is an alternative to classical development by incrementally transforming specifications to code. Here, models represent major artifacts during development. Depending on the type of MDSD approach, however, the role of the models varies considerably. Brown et al. draw the scale from models for more intuitive visualization of code to model-only MDSD setups, where modeling fully replaces classical code-based engineering [33]. A central aspect here is the shift in focus from code-based development of the SUD to development in the modeling tool, which consequently generates the application code. This change to a modeling notation increases the level of abstraction, which ultimately improves productivity due to better understandability and the automation of error-prone steps [155]. However, one should always have in mind the fundamental characteristics of models identified by Stachowiak. [159]

- A model is a mapping from a concrete original to an abstract format
- A model serves a context-specific purpose
- A model represents a simplification, by omitting irrelevant information for the current context

Through different uses of models in MDSD, a number of variants have emerged, which differ in their varying degrees of a strict interpretation of the original idea. The weakest interpretation of the MDSD idea is represented by Model-Based Software Development (MBSD), whereby the models are used as a basis for discussion or for visual support of specification documents. However, the application code is still derived manually from the specification. [37]

Today's use of MDSD in the context of the modeling community corresponds in large parts to the original idea of this development approach. Models replace parts of the classical specification and are used for automated processing. However, model-driven approaches are limited to a specific aspect of development, such as testing. Often, the two concepts MBSD and MDSD are not distinguished clearly. [37]

The strictest interpretation of the original MDSD idea is the Model-Centric Software Development (MCSD). Here, the model artifacts created during development are regarded as the central knowledge base and are linked across the various aspects of development. Furthermore, knowledge gained during development or an automated processing step is usually reflected back into the model(s). [29]

This results in different characteristics of how modeling is used in the development process. Either in sense of visualization of design and architecture decisions or for the purpose of automated translation towards the target platform. Depending on this, the requirements on syntax and semantics as well as the modeling domain guide the decision process on which modeling language fits best. The application domain often decides whether a Domain-Specific Modeling Language (DSML) or a General Purpose Modeling Language (GPML) is used. While DSMLs have limited expressiveness and a strong focus on the particular domain, GPMLs offer a great range of customization options and are widely known [68]. For example, the Unified Modeling Language (UML) is often applied in semi-formal development contexts because of its popularity.

4.1 Meta-Object Facility

Being able to specify a modeling language, the Metamodel (MM) is essential. Clark et al. [44] gives the following definition.

Definition 1 (**Metamodel**)
A metamodel is a model of a language that captures its essential properties and features. These include the language concepts it supports, its textual and/or graphical syntax, and its semantics (what the models and programs written in the language mean and how they behave).

Extending the idea of a MM, the Meta Object Facility (MOF) standard developed by the OMG tries to establish a common modeling layer above different languages, the Metametamodel (MMM). Among others, UML plays a central role, since its modeling concepts are partially used in the context of the MOF MM definitions. The specification bases on a layered architecture which further impacts the field of transformations between different MMs, defined in conformity with the MOF, by Model-to-Model Transformations (M2MTs) (see section 4.2). [129]

The number of layers depends on the problem domain, but usually, four layers are defined. Figure 4.1 shows the subdivision of a MOF-compliant specification into four layers, with the UML and Interface Definition Language (IDL) being placed in the layers here as examples.

M3 is the highest level of the hierarchy and contains the MOF MM. This MM defines the selection of modeling elements for the next lower level. I.e. all MM components of an *M2* model represent instances of the MOF MM elements at the *M3* level. As shown in the figure, the components of the MMs of the UML and IDL represent instances of the MOF MM.

The same applies to the lower levels. For example, the model artifacts on the *M1* level conform to the respective language definition given on the *M2* level. Again talking about the UML example, the models specify the parts of a concrete SUD. Consequently,

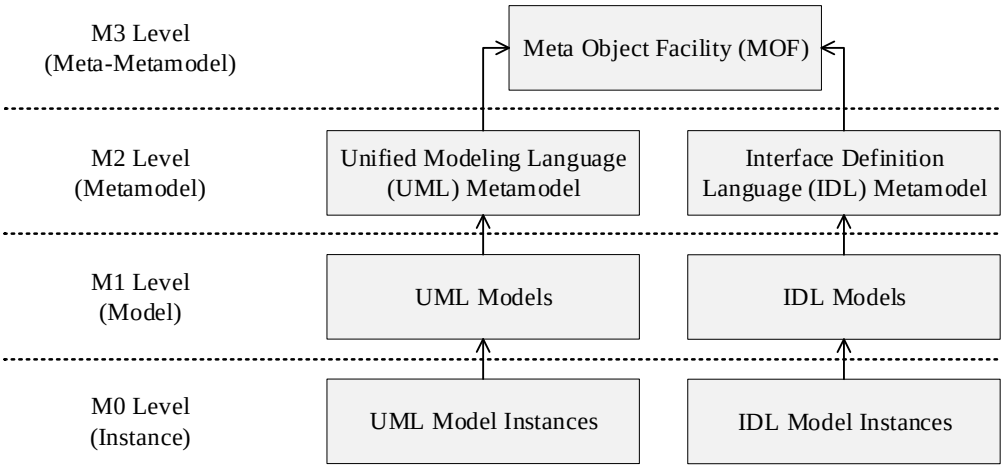


Figure 4.1: Hierarchy of metamodeling in MOF

the artifacts on the lowest level of the hierarchy, namely the *M0* level, represent the objects or instances of the SUD at runtime.

Altogether, the MOF standard and its layer model provide a uniform basis for the definition of modeling languages. Depending on the application, only a subset of the MOF, the so-called Essential MOF (EMOF), can be used for the specification of MMs. However, if the entire language scope is required, it is usually referred to as the Complete MOF (CMOF).

4.2 Model Transformations

As already stated, transformations serve many purposes, especially in MDSD. While there are many flavors of transformations in the context of software development, we narrow down our focus to model transformations [47]. Clearly pointing out the general meaning of transformations, Kleppe et al. [113] provided the following set of definitions, whose correlation is illustrated in figure 4.2.

Definition 2 (**Transformation**)
A transformation is the automatic generation of a target model from a source model, according to a transformation definition.

Definition 3 (**Transformation Definition**)
A transformation definition is a set of transformation rules that together describe how a model in the source language can be transformed into a model in the target language.

Definition 4 (**Transformation Rule**)
A transformation rule is a description of how one or more constructs in the source language can be transformed into one or more constructs in the target language.

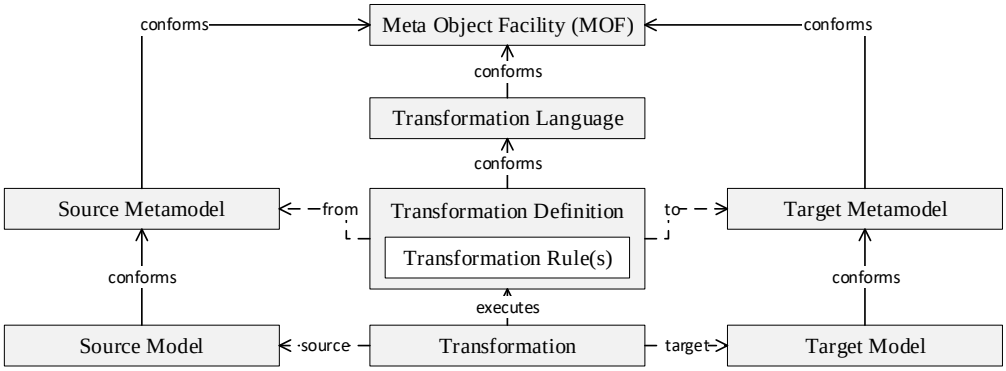


Figure 4.2: Relation of basic concepts in M2MT (based on [56])

However, if we again take a closer look at the M2MT building blocks, certain characteristics of the source and target models plus their combinations allow us to concretize the type of transformation, namely the *combination of source and target MMs*, the *abstraction level of source and target models*, and the *targeted language dimension* [127].

The first one deals with the combination of source and target MMs. In case of the source and target models being specified according to the same MM, an *endogenous* model transformation is taking place. In the context of model transformation, literature commonly uses the term *rephrasing*. In contrast to that, the underlying MMs for source and target may vary, which is widely known as a *translation*. In this case, an *exogenous* model transformation needs to be specified. [171]

In addition to the previous one, the levels of abstraction, the source, and target models are arranged at, reflect another important aspect. If both models, source and target, reside on the same level of abstraction, we speak of a *horizontal transformation*. Examples of this kind of transformation are *refactoring* (endogenous) and *migration* (exogenous) [127]. The counterpart is the *vertical transformation*, whereby the level of abstraction of the model is changed by the transformation.

The third one addresses the language's dimension the transformation primarily aims at. The distinction is made here between *syntax* and *semantic*. While a *syntactical* transformation only cares about the change of the representation of language constructs, the *semantical* transformation cares about the subsequent changes of its meaning. Therefore, semantical transformations are significantly more complex to define and hard to be verified as correct. Taking all the characteristics into account, we end up in a variety of different types of model transformation, which are used in the context of MDSD. Besides the previously mentioned use case of horizontal endogenous transformations in the context of the MOF layer model, vertical exogenous transformations are prominent, when MDA is carried out (see section 4.3).

From a technical point of view, there are many languages and frameworks to apply model transformations in the MDSD context. There are many aspects to include, while determining the transformation approach which fits best. Czarnecki and Helsen [47] categorized the M2MT approaches as follows:

- Direct manipulation
- Structure-driven
- Operational
- Template-based
- Relational
- Graph-transformation-based
- Hybrid

Among others, prominent examples from literature (and partially applied in a commercial context) are the Query View Transformation (QVT) language (part of the MOF standard), the Atlas Transformation Language (ATL), and the VIsual Automated model TRAnsformation (VIATRA) framework [47]. Apart from the VIATRA framework, which is categorized as a *Graph-transformation-based* approach, the mentioned transformation languages represent *hybrid* concepts. For example, the QVT is divided into three components from different categories, *Relations*, *Operational Mappings*, and *Core*. In the context of the A3F (see section 7.3) the Query View Transformation Operational (QVTO) plays a central role for horizontal endogenous M2MTs, where missing semantics is made explicit within the transformation.

4.3 Model-Driven Architecture

In the context of MDSD, another OMG standard, namely Model-Driven Architecture (MDA), comes into play. MDA describes a software development approach that, driven by models and the automation of essential processing steps, should improve the development quality for complex systems. The standard introduces concepts supporting the strict separation of concerns, views, and languages specific to the underlying application domain, which altogether structure the way from a model-based specification down to an executable system and are commonly carried out by UML and thereon defined profiles. Further, the application of vertical exogenous M2MTs and Model-to-Code Transformations (M2CTs) is a core concept of the MDA standard to deal with the layers of abstraction and representation. [67]

In particular, artifacts are differentiated on their level of detail and degree of platform-specific information, as shown in figure 4.3. The elaborations on the different layers of the MDA hierarchy are based on the published specification document [126].

Starting on the most abstract level of the MDA hierarchy, the so-called Computation Independent Models (CIMs) are arranged. The models on this level describe the requirements and needs of the SUD without any implementation details, e.g. user requirements or business objectives. Sometimes, these kinds of models have entitled

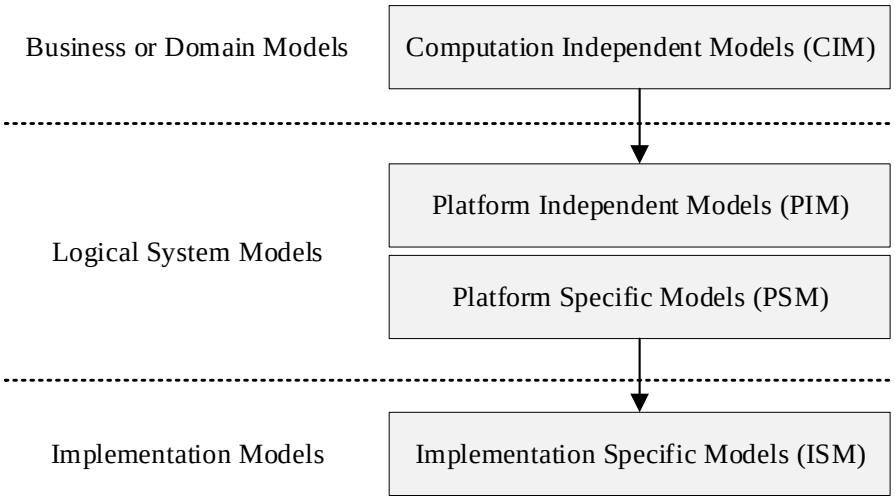


Figure 4.3: MDSD based on the MDA standard

business or domain models. Having specified the CIMs, these models need to be set in relation, further enriched with information and consequently transformed making up the so-called *logical System Models*.

These logical System Models are further divided into two groups, i.e. the Platform Independent Models (PIMs) and the Platform Specific Models (PSMs). What the acronyms already indicated, is the differentiation by means of the included platform properties. Therefore, the PIM defines the behavior of the SUD in terms of stored data and performed algorithms, while no technical details are specified. In contrast, the PSM introduces all the technical knowledge needed to end up with models, which are ready for transformation towards *implementation models*.

The lowest level of the MDA standard holds the Implementation Specific Models (ISMs), commonly created in an automated way. In case of a fully-featured PSM and a high quality (or demonstrably correct) code generator, the code artifacts do not need any human intervention, except for legacy application and subsequent maintenance.

Overall, the MDA approach has many beneficial aspects, e.g. automation or reuse, which build on the division into levels and the strict separation of platform-specific and independent information. However, there are also disadvantages that are mainly originated in the recommendation for the use of UML and the thereon-based tooling. The UML and its profiles for example limit the modeling capabilities and hardens a targeted automated code generation due to the absence of explicit concepts.

5

Verification and Validation in Software Development

Apart from the constructive tasks of a Software Development Life Cycle (SDLC), no matter which concrete flavor is carried out, there is a strong need for process steps constantly evaluating intermediate development artifacts under certain aspects. In literature, these process steps are commonly divided into two types, namely *Verification* and *Validation*. To give an intuitive understanding of the terms, Barry Boehm introduced the following mnemonic [139]:

"Are we building the product right?" ↔ *"Are we building the right product?"*

The first, incorporating the verification of the SUD, aims at analyses assuring the conformance of intermediate artifacts to the specified requirements. Thus, it is a more technical task to be carried out on certain development artifacts. The second, incorporating the validation of the SUD, tackles the analyses checking whether the product meets the customer's needs, initially captured by requirements. Consequently, this kind of task demands knowledge of the application domain and the concrete context. [19]

In order to detail the intuitive understanding of V&V and elaborate on which kinds of defects of the SUD may be revealed, we introduce the well-known fault/failure model [24] [19]. To distinguish various flavors of threats to dependability makes it easier to define the capabilities and limits of concrete V&V approaches.

Definition 5 (**Fault/Failure Model**)
The Fault/Failure Model is made up of the following three definitions:

1. ***Fault***
   A fault is given by a static defect in a system's source code. Further, a fault is classified as dormant *in case of the affected code fragment not being executed and* activated*, if it is processed during execution.*
2. ***Error***
   An error represents a diverging internal state, as a result of a fault being activated.
3. ***Failure***
   A failure is an observable deviation between the actual and the expected behavior of a system.

The three concepts show the stages of activation/propagation a defect may be in. This further draws the spectrum for V&V approaches, ranging from static analyses of intermediate results to dynamic approaches requiring executable development artifacts.

On the one hand, the field of static analyses ranges from quite informal and manual techniques like reviews to pretty formal verification approaches, e.g. model checking. Strongly depending on the combination of the applied technique and the type of development artifact under analysis, the resulting quality statement is more or less concrete and error-prone. A common feature is the fault-centric reporting of identified issues.

On the other hand, approaches in the field of dynamic analyses just can draw statements about observed failures. In general, there are informal/manual approaches, e.g. inspection, and more automated approaches like simulation or interpretation of developed functionality.

As a mixture of both kinds, verification, and validation, there is a quite popular approach for quality control during ongoing development, namely *Software Testing*.

5.1 Software Testing

In general Software Testing has many different flavors and therefore lots of definitions. The International Software Testing Qualification Board (ISTQB) defined testing as follows [80]:

Definition 6 (**Testing**)
The process consisting of all life cycle activities that deal with the planning, preparation, and evaluation of a software product and its associated work results. The aim of the process is to ensure that these meet all specified requirements, that they fulfill their purpose, and that any error conditions are detected.

Furthermore, two types of software testing are distinguished. The so-called *static tests* are techniques, which may be performed on each and every development artifact. Here, mainly structured variants of inspection, e.g. reviews, are performed to assure quality, which most of the time remains a purely manual activity. In contrast, the *dynamic tests* are based on executable development artifacts. During the ongoing execution of these artifacts, the steady observation of the SUD in a structured way aims at revealing failures. [174]

In contrast to other V&V techniques aiming at the proven correctness of the SUT, limitations of testing should be kept in mind.

> *"Program testing can be used to show the presence of bugs, but never to show their absence."* - Edsger W. Dijkstra [58]

Thereby, Dijkstra tackles a common misunderstanding. Testing marks an approach for improving the user's trust in the absence of defects/bugs as well as in the software shipping with the intended functionality with regard to the customer's requirements, but may not prove the SUT correct. In addition to this central principle of testing, there are lots of basic principles apart from the one quoted by Dijkstra, guiding testers and developers towards a sufficient realization of testing throughout the product life cycle.

5.1.1 Fundamentals of Testing

In addition to the insight on the *missing ability for showing the absence of defects*, testing may *not be carried out exhaustively*. Further, there is a strong need for an *early application* and tight focus of the testing activities, in order to reduce the overall costs for testing. Especially in late phases of testing, e.g. in a pre-release phase, a tester should always keep in mind the phenomena of *defect clustering*, which says that most of the remaining defects are encapsulated in a small number of modules of the SUD. In a row with the *pesticide paradox*, which complains about the repeated application of the same test suite and its expressiveness over time, these principles are based on experiences over the past decades of testing software. Finally, *testing is heavily context-dependent* and therefore needs to be adapted according to requirements imposed by the development context as well as regulatory instances. Further continuing the thought of the previous section, testing is not a pure verification nor a pure validation technique and therefore does not give any causal relation between the absence of faults in the product and its usability for the customer. [80]

All the presented insights on testing commonly lead to a specific variant of how testing is carried out during development. However, all different variants follow the same basic structure, shown in figure 5.1 and commonly known as the Software Testing Life Cycle (STLC). Although the figure shows a linear process, the linearity is not a necessity, i.e. certain steps of the process may either overlap or be iterated during the process.

Throughout the following sections, we detail activities included in the process steps, while introducing important definitions for the field of software testing. The contents detailed in these sections are based on the widely accepted definitions of testing activities from the ISTQB [80] [174].

Test planning and control

As illustrated in figure 5.1, the first step of the STLC deals with planning and control activities. In particular, *Test Planning* aims at understanding the goals and objectives of the customer. Furthermore, all the involved stakeholders need to be addressed. Overall, the project context needs to be identified and understood in detail. For a test engineer, this results in a *test mission*, sometimes called a *test assignment*.

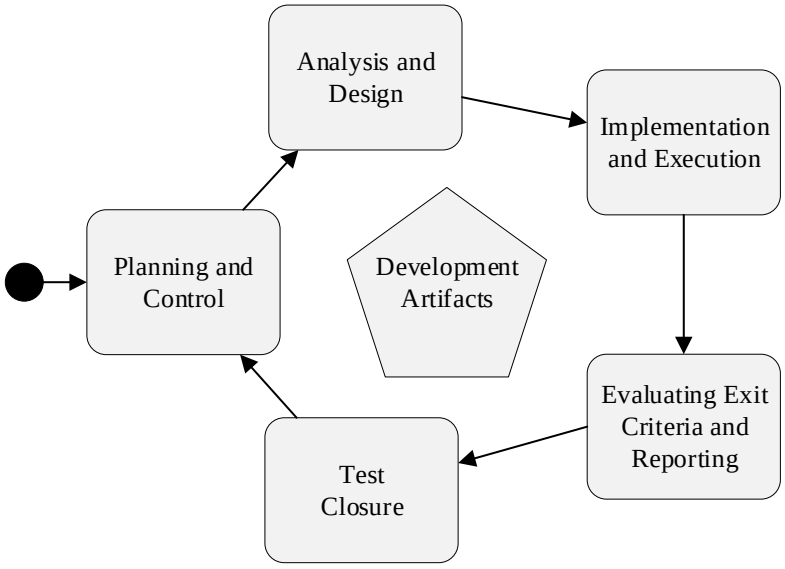


Figure 5.1: The common software testing life cycle

Based on this intermediate version of a test mission, a conformity check with the companies *test policy* as well as the overall *test strategy* needs to be performed. In case of conformance issues, the affected parts of the test mission are adjusted.

Definition 7 (**Test Policy** [170])
A high-level document describing the principles, approach, and major objectives of the organization regarding testing.

Definition 8 (**Test Strategy** [170])
A documentation aligned with the test policy that describes the generic requirements for testing and details how to perform testing within an organization.

As soon as the general planning is finished, the basic testing approach and therefore necessary resources are determined. The range of available approaches is usually divided into functional, non-functional/quality, structural, and regression testing approaches. The first two categories are sometimes subsumed under the term *Black-box Testing*, while structural tests are often called *White-Box Testing* approaches. Apart from the mentioned non-regression categorizations, sometimes there is an additional category called *Grey-Box Testing*, reflecting partial or uncertain knowledge of the SUT.

Definition 9 (**Black-box Testing** [170])
Testing, either functional or non-functional, without reference to the internal structure of the component or system.

Definition 10 (**White-box Testing** [170])
Testing based on an analysis of the internal structure of the component or system.

One aspect that is closely related to the chosen test approach is the definition of an *exit criterion*. This criterion determines whether testing activities are completed or not. Completing the set of test planning activities, the subsequent process steps as well as included tasks are scheduled according to the project timeline.

In contrast to the planning part, the *Test Control* activities are usually carried out in parallel to all remaining process steps included in figure 5.1. At this point, the structure of the figure seems misleading, but above all tackles the definition of how the control activities are performed. An essential aspect of the test control process step is the steady comparison of the actual testing progress against the planned progress. This is followed by a continuous reporting of the derived status to all the previously identified stakeholders. In case of unacceptable deviations from the plan, an adjustment of the plan based on the monitoring information is triggered. Two types of triggers are conceivable. On the one hand, corrective actions due to misconceptions in the planning phase, e.g. change of exit criteria. On the other hand, expanding actions through a more detailed view of the SUT.

Test analysis and design

Based on the developed test plan, test engineers further detail the intended test setup by analyzing artifacts and thereby developing a concrete design. In particular, development artifacts like requirements, architecture documents, design specifications, and included interfaces, also known as the *test basis*, are studied. At this point, the test basis serves as sufficient information for the design of black-box *test cases*. Even if no black-box tests are intended, the *test conditions* for the relevant *test items* are developed in this phase of the STLC. Here, especially the testability of the underlying test basis is challenged and, if necessary, triggers a change request for the test basis. A central role of the test design phase is to end up with a focused set of tests for the targeted parts of the SUT.

Definition 11 (**Test Condition** [170])
A testable aspect of a component or system identified as a basis for testing.

Definition 12 (**Test Item** [170])
A part of a test object used in the test process.

Definition 13 (**Test Case** [170])
A set of preconditions, inputs, actions (where applicable), expected results and postconditions, developed based on test conditions.

Besides the analysis and design of test cases, the *test environment* marks an essential point of investigation in this phase. On the one hand, the necessary infrastructure to perform subsequent test phases is determined. On the other hand, appropriate tooling

for the test engineer is set up and configured to integrate to a sufficient toolchain that operates on the previously defined infrastructure. Overall, the goal of this STLC-phase is a tangible set of test conditions and test procedures with regard to the predominant set of test objectives.

Definition 14 (**Test Environment** [170])
An environment containing hardware, instrumentation, simulators, software tools, and other support elements needed to conduct a test.

Test implementation and execution

Following the analysis and design view on testing, the STLC-phase of this section deals with the concrete *implementation of tests* and the thereon-based execution. In particular, during *test implementation*, the previously specified set of *abstract test cases* is detailed by developing appropriate test data to derive a set of concrete test cases. Moreover, test procedures are implemented by scripts to support the execution of tests in the test environment. In order to make the execution more efficient, the test cases are grouped into logical collections, called *test suites*, sharing test data or test objectives. Furthermore, the test suites are prioritized and a schedule for the subsequent execution is derived. Any missing components of the test environment are implemented and verified in this phase to create the best possible starting point for the following phases.

Definition 15 (**Abstract Test Case** [170])
A test case without concrete values for input data and expected results.

Definition 16 (**Test Suite** [170])
A set of test scripts or test procedures to be executed in a specific test run.

Following the implementation of test-related artifacts, the *test execution* is performed. In particular, the previously defined test suites are executed against the SUT. The recording of the execution traces and results is the most important task, since based on these so-called *test logs*, documents or even instructions for action are derived for stakeholders. Especially in cases, where observed behavior differs from intended behavior, the test log marks the basis of the decision to develop appropriate countermeasures.

Definition 17 (**Test Log** [170])
A chronological record of relevant details about the execution of tests.

Evaluating exit criteria and reporting

Continuing the STLC presented in figure 5.1, an important point of decision in the process is reached. As its name implies, the *evaluation of exit criteria* determines whether

sufficient testing has been performed. Two different conclusions can be drawn from this check. Either that the previous tests were not sufficient and further iterations are necessary, or that the criterion chosen in the planning phase is not suitable and must be adapted. This step applies to each of the defined *test levels* from *Unit/Component* testing up to *Acceptance* testing. Regardless of the current test level, a *test report* is generated, which discloses the current testing status for stakeholders.

Definition 18 (**Test Level** [174])
Depending on the constructive phases for the test basis, the test levels are commonly divided into the following levels: unit/component testing, integration testing, system testing, and acceptance testing. A more fine-grained division into levels is conceivable if required by the test policy or the product.

Definition 19 (**Test Report** [170])
Documentation summarizing test activities and results.

Test closure activities

If the exit criteria take effect and these are considered useful, the final actions of the STLC, namely *test closure activities*, are performed. The primary objective is to compare planned test artifacts with artifacts that have been created. Furthermore, this includes a review of recorded problems and knowingly remaining errors to ensure appropriate defect handling and documentation.

Apart from test result-specific closure activities, some test environment-related tasks need to be performed. This includes a handover of test-related artifacts to the maintenance department as well as archiving all the test tools and infrastructure relevant for the previously detailed process steps. Moreover, improvements for future variants of the STLC are identified at this point and necessary adjustments are documented and initiated.

As already mentioned in the introductory part of this section, the concrete instance of a STLC is subject to many influences such as the type of STLC, the companies test policy or general strategy, empirical values, and standards set by the application context.

5.1.2 Standardization

Apart from ISTQB's view on the software testing process, there are many other standardization bodies, ranging from international boards like International Organization for Standardization (ISO), International Electrotechnical Commission (IEC), or Institute of Electrical and Electronics Engineers (IEEE), to national organizations like Software Engineering Institute (SEI), American National Standards Institute (ANSI), or British

Standards Institute (BSI). Over the past decades, all the mentioned boards competed by publishing standards for software quality assurance and control. Figure 5.2 gives a high-level overview of the most impacting standards, some of which are no longer active at this time or have already been summarized in new standards.

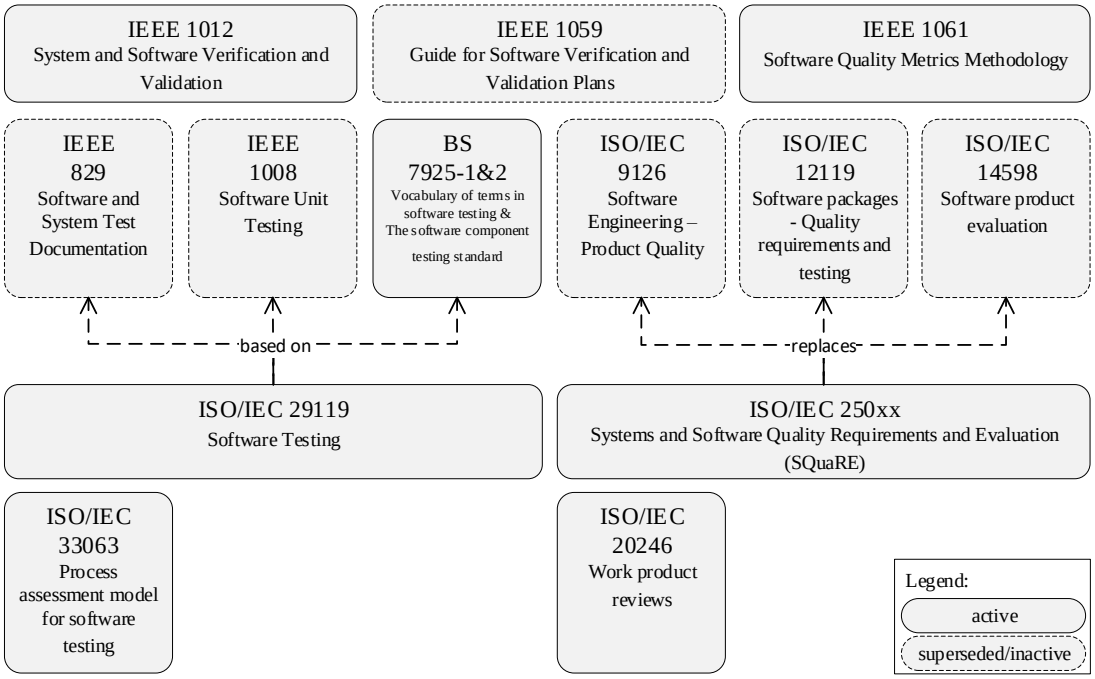


Figure 5.2: Important standards in the context of Software Testing and Quality Assurance

The first group of standards arranged in the upper part of figure 5.2, deals with general topics on V&V of systems/software and gives advice in terms of the applied methodology as well as involved artifacts. Over the years, some standards were superseded by newer revisions, in order to continuously align the contents to recent insights, while others were set inactive or even withdrawn. The second group placed in the center of figure 5.2 includes standards in the area of testing. Furthermore, the objects placed on the left side represent basic and relatively new standards from the field of dynamic software testing. This is contrasted by the objects on the right-hand side, which represent the quality assurance domain and its development over the years. At the bottom of figure 5.2 two more standards are mentioned, which are often used in conjunction with *ISO/IEC 29119* and *ISO/IEC 25000*.

Overall, attempts have been made in recent years to consolidate the almost unmanageable number of standards. In particular, the discussed standard *ISO/IEC 29119* attempts to summarize and update established standards and prescribe the basic processes across the different development practices.

ISO/IEC 29119 - Software Testing

Based on the contents published in [146], this section gives a deeper insight into the controversially discussed standard. *ISO/IEC 29119* was designed with the aim of specifying a standard that combines best practices and reflects the state of the art of software testing from a process-level view. As mentioned above, this standard contains elements of older standards in this field and brings them together in a larger context. For a better structuring of the contents, it consists of five separate parts.

ISO/IEC 29119-1 [6] contains mainly the definitions and concepts, which form the basis for the following parts and are mostly based on the contents of *BS 7925-1* [40]. *ISO/IEC 29119-2* [7] on the other hand presents the contents of an intended and potential reference process for software testing continuing the concepts of *BS 7925-2* [41] and *IEEE 1008* [21]. Furthermore, this part puts a focus on the cross relations between formerly separated process groups, which are examined in more detail below. Overall, this part reflects the major contribution of this standard. *ISO/IEC 29119-3* [8] focuses on documentation topics, which mark an essential aspect of a meaningful software testing process. In particular, the necessary documentation parts, as well as their purpose within the previously mentioned reference process, are presented. Moreover, the general concepts for the technical documentation are detailed. *ISO/IEC 29119-4* [9] investigates applicable techniques for test design and implementation activities in software testing. Starting from the techniques, which were presented in the context of *BS 7925-2*, alternative possibilities are listed and in the further course also explicitly dealt with coverage criteria. *ISO/IEC 29119-5* [10], the last part of the standard, elaborates on a special testing technique, namely keyword-driven testing. In addition, further concepts of this technique, such as hierarchical keywords and templates for technical keywords, are discussed and a possible application in the presented reference process is shown.

Apart from the division into five parts, the standard aligns to a layered structure of process groups with mutual dependencies as per figure 5.3.

Organizational Test Process The highest level of processes included in the layered structure presented in figure 5.3, deals with the *organizational test process*. Apart from a concrete project, a process template is given, which among others enables to manage and develop the organizations' test policy and strategy. Therefore, the template is split into three main activities, namely *Development of Organizational Test Specification (OT1)*, *Monitor And Control Use Of Organizational Test Specification (OT2)*, and *Update Organizational Test Specification (OT3)*. *OT1* represents the starting point for the creation of an Organizational Test Specification (OTS), while *OT2* and *OT3* support the iterative improvement and maintenance of related artifacts.

Test Management Processes Below the organizational test process, project-specific test management processes are arranged. These in turn are divided into three groups (see figure 5.3).

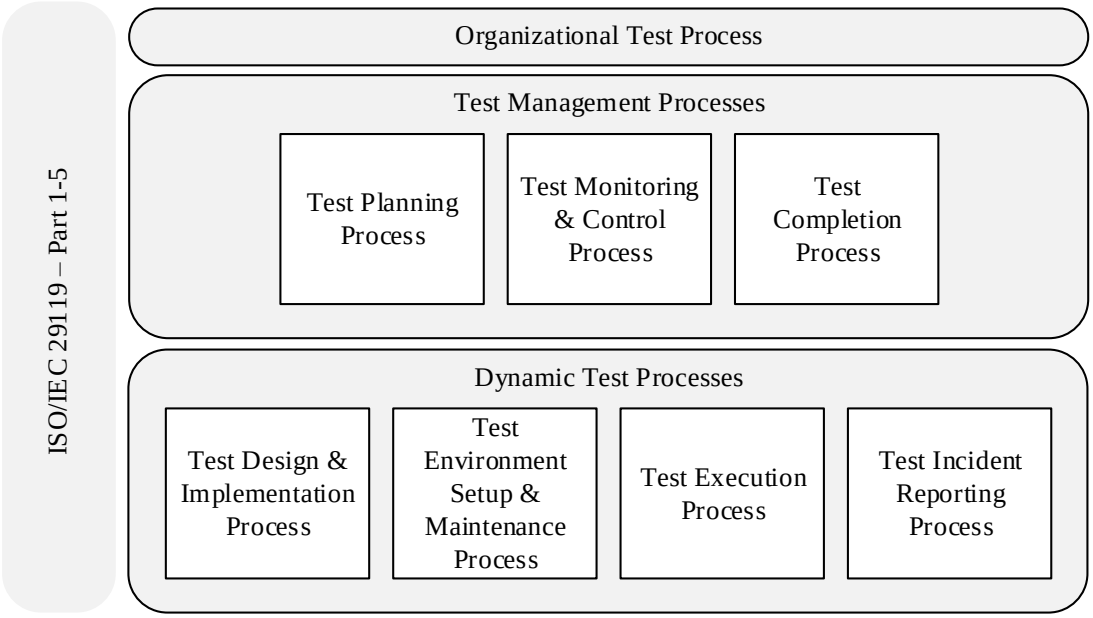


Figure 5.3: Process Structure defined in ISO/IEC 29119 (based on [146])

Test Planning Process - As an initial activity, the context is first assessed, the workload is estimated and further allocated (TP1 - TP2). Following the initial activities, the identification, mitigation, and deduction towards a risk-driven strategy mark essential tasks (TP3 - TP6). At this point, it is recommended in the standard to iterate the steps several times to achieve improved quality and focussedness of the resulting test plan. Closing activities tackle the documentation and communication of developed artifacts in the context of the test plan (TP7 - TP9).

Test Monitoring & Control Process - Based on the derived test plan, this process deals with the continuous monitoring and control of subsequent testing activities. Therefore, monitoring infrastructure is deployed accordingly (TMC1). Based on a previously determined set of test measures the monitoring is performed (TMC2) and in case of deviations triggers specified control structures (TMC3). All the occurring incidents are collected and in a final activity shipped in detailed reports (TMC4), which further marks the basis for decisions about sufficient testing.

Test Completion Process - The last of three test management processes takes place if the test monitoring and control processes determined the end of testing activities. The final activities incorporate the archiving of test assets (TC1), a test environment cleanup (TC2), as well as reflective and again documenting tasks (TC3 and TC4). Thereby, the last two activities pursue the goal of continuous improvement of the process landscape.

Dynamic Test Processes Apart from the previously presented management processes, but tightly coupled, the processes dealing with the concrete testing activities arranged at the bottom of figure 5.3, are furthermore detailed.

Test Design & Implementation Process - Before the execution of test cases may be carried out, the design and implementation of those need to be completed. With regard to the actual test plan, the concrete design, and implementation of test cases are developed iterative (TD1 - TD6). Overall, there are three different artifacts, which specify the results of respective activities:

- Test Design Specification
- Test Case Specification
- Test Procedure Specification

All these artifacts contain essential information as soon as the execution of test cases is carried out.

Test Environment Setup & Maintenance Process - Completing the initial test environment setup, the activities of this process ensure the ability to execute test cases by continuous maintenance of the environment (ES1 and ES2).

Test Execution Process - Carrying out the major task of testing, this process defines three activities. First, the procedures included in the test procedure specification are executed (TE1). The results are further evaluated (TE2) and finally documented, which again serves the overall goal of steadily improving testing and test-related processes.

Test Incident Reporting Process - The last process of the dynamic test processes group handles unforeseen incidents occurring during test execution. In particular, two steps constitute this process, the former analyzing the incidents (IR1) and the latter documenting these incidents with regard to the affected stakeholders.

Relation between ISO/IEC 29119 and ISTQB

Although ISTQB is not a standardization body, the documents and materials in the context of their certification programs are seen as standard-like. This is due to the community-oriented development and practical relevance of the material, reflecting a wide-accepted view on the topics. As we have seen in the previous sections, the *ISO/IEC 29119 standard* and the *ISTQB concepts* differ only slightly. This can be attributed to the fact that both variants are based on an overlapping set of documents and thus logically continue these approaches in large parts. Nevertheless, the ISTQB view seems to leave more room for specific application contexts and test methods, which was one major point of criticism when promoting the publication of the *ISO/IEC 29119* standard. Even petitions against the publication of this standard were started, fearing big players of the industry behind the scenes forcing the conformance to this standard in future projects, thereby hindering appropriate approaches. Overall, the two documents draw a very similar picture of how software testing should be carried out.

5.1.3 Test Design Techniques

As we have seen in section 5.1.1, the design and implementation of test cases play a major role. There are several fundamentally different approaches to derive the most efficient and targeted set of test cases from an existing test basis. As already mentioned, a distinction is usually made between *Back-box/Specification-based approaches* and *White-box/Structure-based approaches* [174]. The ISTQB introduces two additional categories of testing techniques overarching the mentioned [170]. On the one hand *Experience-based Test Techniques* and on the other hand *Defect-based Test Techniques*. To give the reader more insight, some representative techniques of these categories are detailed. The selection of example technologies refers to the test design techniques considered in our approach and thus represents the basis for later understanding.

Experience-Based Approaches for test design are not primarily based on development artifacts. Rather, the essential components of these techniques are the knowledge, skills, and background of the testers. As its name implies, the experience in similar systems from different perspectives (technical vs. business) enables the testing team to guess *what may go wrong* and *what needs to be covered* by a comprehensive set of test cases. A simple technique that extensively makes use of intuition *what might go wrong* is called *Error Guessing*. Here, possible error cases are collected and subsequently transformed into a set of test cases to confirm the assumptions. [80]

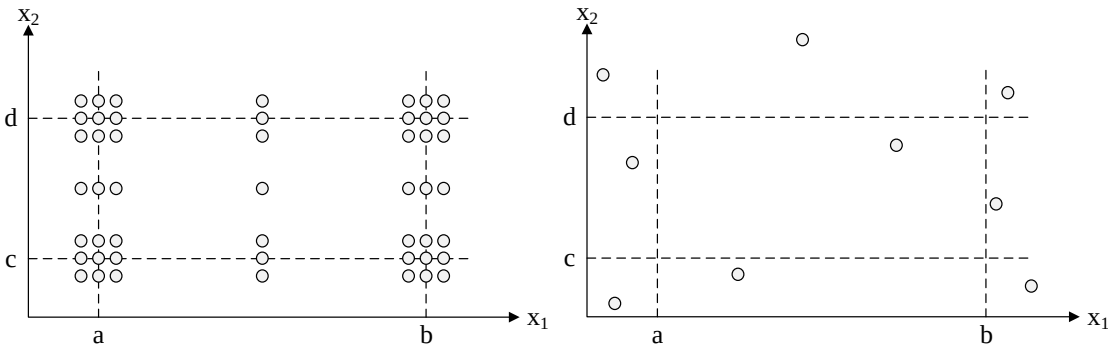
Besides the *Error Guessing* approach, there is another technique in this field called *Exploratory Testing* or *Ad-hoc Testing*. Driven by a group of testers, the *Context-Driven School*, the following compact but meaningful definition emerged.

"*Exploratory testing is simultaneous learning, test design, and test execution.*" - James Bach [25]

Bach further pointed out, that each flavor of testing is *exploratory* to some degree [25]. In contrast to scripted testing, the continuous learning about the SUT by developing new test cases based on insights of previous test runs marks the core concept. This approach does not put the *obvious* parts of the system to the test but may investigate cases that would not be achieved by non-exploratory techniques. Therefore, *Exploratory Testing* is not just a classical test design technique, but a way of thinking about testing. [25]

Specification-based Approaches take the requirement or similar specification artifacts to design test cases in a structured way. Therefore, the category is called *Black-box*, which is due to the missing information about the internal behavior of the SUT. However, based on this limited amount of information, characteristic and error-prone combinations of inputs are determined, which further serve as stimuli to the SUT. [174]

Examples from this area are *decision tables*, *state transition testing*, *Boundary Value Testing*, and *Equivalence Class Testing*, the latter two techniques being discussed in more detail below. [80]



(a) Boundary Value Testing (b) Equivalence Class Testing

Figure 5.4: Specification-Based Test Design approaches (subfigures based on [103])

Boundary Value Testing, sometimes called *Boundary Value Analysis*, marks a test design technique, which focuses on interface information of a component or system under test. Therefore, the set of input data is divided into its atomic elements. Each of these data elements defines a scale of possible values given by its datatype. This scale may further be partitioned into sections based on specified thresholds or constraints defined for the data elements. What *Boundary Value Testing* aims at is selecting test data in a structured way by data points at the boundaries of the determined sections. To demonstrate this approach, figure 5.4a shows a tiny example, where the axes represent the scale of variable x_1 with boundaries a and b and the variable x_2 with boundaries c and d. Each of the gray dots included in this figure represents a test data point that assigns values to the variables x_1 and x_2. The amount of test data, in this case, is determined by a specific variant of *Boundary Value Testing*, namely *Robust Worst-Case Boundary Value Testing*. Apart from this specific variant of *Boundary Value Testing*, there are lots of other less strict flavors, leading to fewer test cases. [103]

Another technique, namely *Equivalence Class Testing*, is illustrated by figure 5.4b on the same example scenario. The limits of variable values imposed by thresholds or constraints define so-called equivalence classes on them. The resulting set of equivalence class combinations determines the value ranges, for which representatives are determined either randomly or according to a certain scheme. The example shown in figure 5.4b again reflects a special version of *Equivalence Class Testing*, namely *Strong Robust Equivalence Class Testing*. For this technique, there are other variants that may fit better to a certain test context and effectively end up with a lower number of resulting test cases. [103]

Particularly, figure 5.4a shows that even with very simple problems, the resulting number of test cases becomes very large. Due to time and budget restrictions, it is necessary to sort the test cases in descending order of importance (*Prioritization*) or to make a selection (*Selection/Reduction*). In both cases, this is often achieved utilizing so-called Risk-Based Testing (RBT). In general, a risk represents a factor that could result in fu-

ture negative consequences and is usually expressed by its likelihood and impact [170]. Based on a risk management process, Amland described RBT as a suitable integration of this process into the conventional test process [18]. The resulting process behind RBT breaks down into the following steps.

1. Planning (risk identification/risk strategy)
2. Identify risk indicators (risk assessment)
3. Identify the cost of a fault (risk assessment)
4. Identify critical elements (risk assessment)
5. Test Execution (risk mitigation)
6. Estimate to complete (risk reporting and prediction)

Besides the application of RBT in the context of specification-based test design approaches, the area of structure-based test design approaches offers even more potential. [65] All in all, RBT offers a mechanism with high practical relevance, which introduces information from other development domains into the testing domain, but itself does not represent a complete test design technique.

Structure-based Approaches, sometimes called *Black-box Approaches*, offer a much more targeted way of deriving test cases from an appropriate development artifact. In particular, these approaches are a necessary supplement to the above-mentioned Black-box approaches, as the accessibility and meaningfulness of certain parts of the respective implementation, are closely examined. [174]

Many development artifacts serve as a sound basis for the application of structure-based approaches to derive test cases. Although mostly code artifacts are used for the application of structure-based approaches, there are lots of other sufficient behavioral descriptions. The underlying graph structure represents the crucial information to which the structural criteria are applied. For this reason, the following definitions always reside on the underlying directed graph structure and abstract from the concrete artifact representation together with its additional data. [19]

For this purpose, we first define a graph.

Definition 20 (**Graph** [19])
A graph formally is made up of

- *a set N of nodes*
- *a set N_0 of initial nodes, where $N_0 \subseteq N$*
- *a set N_f of final nodes, where $N_f \subseteq N$*
- *a set E of edges, where E is a subset of $N \times N$*

Based on the set of nodes and directed edges, possible ways through the graph result, which represent valid (partial) executions of the system specified by the graph. In the

case of source code artifacts, the graph usually reflects the control flow of the program. A sequence of nodes and edges through a graph is further called a *path*.

Definition 21 (**Path** [19])
A path through a graph is a sequence $[n_1, n_2, \ldots, n_M]$ *of nodes, where each pair of adjacent nodes,* $(n_i, n_{i+1}), 1 \leq i \leq M$*, is in the set E of edges.*

Furthermore, if the path complies with some characteristics regarding the start and end nodes, it is called a *test path*.

Definition 22 (**Test Path** [19])
A path p, possibly of length zero, that starts at some node in N_0 *and ends at some node in* N_f*.*

These test paths in turn represent the runs through the graph, which are triggered by test data of the corresponding test case. For deterministic systems represented by a graph structure, the mapping between test data and test paths follows a many-to-one relation [19]. Based on these definitions, the term coverage regarding the graph structure can be defined as follows.

Definition 23 (**Graph Coverage** [19])
Given a set TR of test requirements for a graph criterion C, a test set T satisfies C on graph G if and only if for every test requirement tr in TR, there is at least one test path p in path(T) such that p meets tr.

The amount, combination, and type of test requirements can of course vary, which results in different variations of coverage.

Structural Coverage Criteria focus on the underlying graph structure and leave aside consideration of concrete data. Such criteria are often used at the unit level in particular. Standards such as *ISO 26262*, *DO-178B/C*, and *IEC 62304* refer to representatives of the structural criteria in its recommendations regarding sufficient coverage of the SUD [5] [2] [11]. Based on the definitions previously introduced, we further specify the widely-known *C0-C2 criteria* [174].

Definition 24 (**Node Coverage (C0)** [19])
Test set T satisfies node coverage on graph G if and only if for every syntactically reachable node n in N, there is some path p in path(T) such that p visits n.

Definition 25 (**Edge Coverage (C1)** [19])
Test set T satisfies edge coverage on graph G if and only if for every edge e in E, there is some path p in path(T) such that p visits e.

Definition 26 (**Path Coverage (C2)** [19])
Test set T satisfies path coverage on graph G if and only if every path(T) equals the set of paths in G.

Besides the three criteria defined, there are several other structure-based criteria, e.g. Modified Condition/Decision Coverage (MC/DC), ranging on intermediate levels according to their strength. Some of them are part of figure 5.5 and thereby set into relation (detailed at the end of this section).

Data Flow Coverage Criteria in contrast to the *Structural Coverage Criteria* consider the use of variables besides. Research in this area goes back to the early 80s, where Laski et al. elaborated on various versions of data flow-related coverage criteria [117]. The consideration of the application context of a variable is of particular importance here and therefore distinguished as follows.

- **Definition**
  A concrete value is bound to a variable (*def*).
- **Usage**
  The value of the variable is used to decide about the subsequent execution flow (*p-use*) or is part of a computation (*c-use*)

Based on these different contexts and to make the definitions of this area consistent with the previous ones, the original definition of a graph is extended as follows.

Definition 27 (**Graph (Extended)** [19])
In the context of data flow investigations, the existing definition of a graph further consists of

- *a set V of variables associated with the program artifacts embedded in the graph*
- *each node n and edge e defines a certain subset of V, with the sets being called def(n) and def(e)*
- *each node n and edge e uses a certain subset of V, with the sets being called use(n) and use(e)*

Consequently, each element of the graph is processed and data sets for the *def* and *use* are initialized accordingly. Analogous to the definitions of structural criteria, predicates (test requirements) can be specified on these sets, from which test cases can be further derived. We do not further elaborate on data flow criteria and leave the investigation to the reader, but we recommend the work of Ammann and Offutt [19].

What is of great interest are the relationships between the individual coverage criteria previously presented. These relationships are often described by so-called *subsumption*, which results in a dependency structure that can be seen in figure 5.5 for some selected criteria. In particular, the criteria with a dotted frame are based on data flow information, while the ones with a dashed frame take the structural characteristics into account. In case of an arrow from criterion A to criterion B is included, A *subsumes* B, i.e. the test cases derived concerning A satisfy B, but not the other way round.

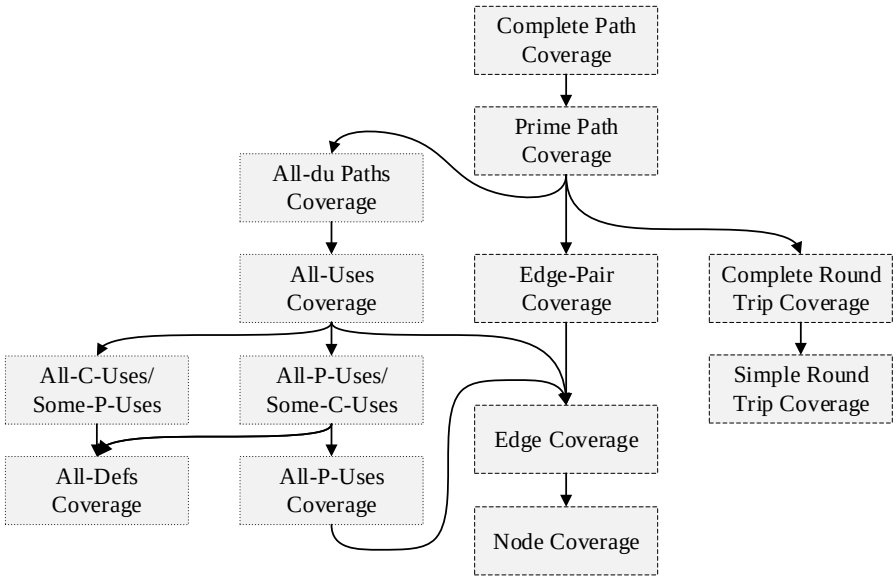


Figure 5.5: Subsumption hierarchy of structural and data flow criteria (based on [19] and [149]

Overall, such a hierarchy of coverage criteria enables developers and testers to find an appropriate criterion, which is due to its expressiveness being set into relation.

Defect-based Approaches in contrast to error guessing, which is based on the experience of the development and testing team, these techniques are thought to evoke the concrete error case and examine the behavior in the context of test case execution. For instance, fault injection or mutation testing are the most prominent examples of this category. *Fault injection* for example leaves the test object in its original state and changes the memory in which the variable values are stored during runtime [93]. Thereby, primarily the robustness of the software, in case of external influences like cosmic rays, is evaluated.

In contrast, *Mutation Testing* or *Mutation Analysis* aims at revealing common coding faults introduced by programmers. Originally this technique was proposed in 1978 by DeMillo et al. [53], who presented a system that made it possible to determine the significance of a test suite. To meet this requirement, the following fundamental hypotheses apply, as summed up by Jia et al. [100].

- **Competent Programmer Hypothesis (CPH) [53]**
  Programmers are competent and therefore tend to create code that is close to a fault-free version. In case of the presence of faults in the final code artifact, these faults are just simple faults, that can be fixed by small adjustments to the syntax. This is why mutation testing focuses on small changes to the underlying syntax, to mimic a competent programmer.
- **Coupling Effect (CE) [53]**
  In contrast to the CPH, the CE aims at types of faults used in the context of mu-

tation. DeMillo et al. formulated it as follows: "Test data that distinguishes all programs differing from a correct one by only simple errors is so sensitive that it also implicitly distinguishes more complex errors."[53] As a result, mutation analysis is limited to the simulation of simple errors, because even complex errors are only a combination of simple errors.

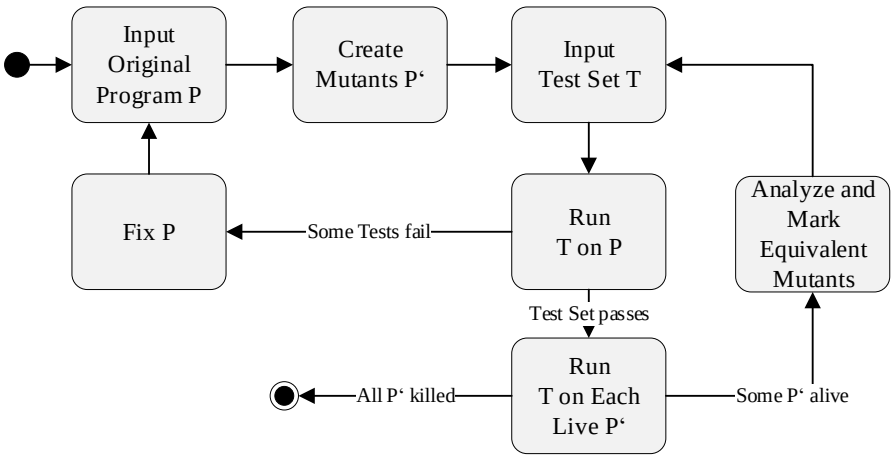


Figure 5.6: Mutation Analysis process (based on [134])

Based on these hypotheses, the process behind mutation testing looks like illustrated in figure 5.6. Starting with the original version of program P, the basis for subsequent mutation testing specific process steps is given. As a first step, the faulty versions of the original program P are created. These faulty versions are usually called *mutants* and represent the products of a technical realization of the previously introduced hypotheses. Thereby, the simple faults are realized by so-called *mutation operators*, which each introduce one type of fault into the program by controlled manipulation of a defined subset of program fragments. Besides, the set of mutation operators is highly dependent on the chosen representation of the program, thus there is no one set of operators, applicable to every problem domain. To create representative and effective mutants, there are further multiple strategies to apply mutation operators and/or even combine their application.

However, as soon as the set of mutants P' is created, the set of test cases T represents the central artifact of the mutation testing process. Therefore, all the test cases included in T are run against the original program P. This step is necessary to determine a valid starting point for subsequent conclusions drawn based on this test set. If any of the tests included result in a test verdict apart from `PASSED`, the program needs to be fixed to lead to a fully passing set of test cases. If all test cases are rated as `PASSED`, the set of test cases is passed to the next process step. Here, all of the test cases included are run against each of the previously created mutants P'. At this point, special wording regarding the relation of mutants and resulting test verdicts is introduced. In case of a test case still being rated `PASSED` at the end of a run against a mutant, we say the mutant is *alive*. All verdicts apart from `PASSED` lead to a mutant classification *killed*. In case of all mutants P' being killed by the test set, the mutation testing process ends.

Otherwise, the test results across all mutants P' are analyzed to conclude the test set. In particular, the Mutation Adequacy Score (MAS) is calculated for this purpose, which provides information on how high the proportion of *killed* mutants is.

Definition 28 (**Mutation (Adequacy) Score**)
The Mutation Adequacy Score (MAS) is defined as

$$MAS = \frac{D}{M - E}$$

with

- D*: number of killed mutants*
- M*: number of created mutants*
- E*: number of equivalent mutants*

One difficulty of mutation testing is explicitly dealt with here, namely mutants that are semantically indistinguishable from the original, so-called *equivalent mutants*. Overall, this score indicates the test set's ability to detect mutants of the original version and mostly serves as a trigger for subsequent modifications of the test set.

However, this imitation of programming errors and the mutation testing process that builds on it is accompanied by some challenges. On the one hand, the cost of performing all tests against all created mutants is the biggest obstacle. To tackle this challenge, many techniques to optimize process steps of mutation testing have been proposed, which is largely covered by Jia et al. [100]. On the other hand, there are conceptual challenges while performing mutation testing. The detection of *equivalent mutants* [42] as well as the *human oracle problem* [172], which is a problem of any test procedure, sometimes still requires human intervention or the use of complex procedures, to enable the practical application. [100]

Overall, Mutation Testing can be used in two different ways for testing. Firstly, as a test design technique in which new cases to be tested are identified by mutations of the original program, from which new test cases are then created. Secondly, as a technique for determining the quality of a test set by constantly observing the MAS. Regardless of the concrete characteristics, defect-based approaches represent a group of practical techniques with high potential, which, however, due to the costs, are not yet widely used in a practical context.

5.2 Model-Based Testing

As already explained in chapter 4, modeling is often used in complex development contexts. This serves above all to abstract irrelevant information for the current use case, as well as the subsequent automation of further manual process steps. These advantages

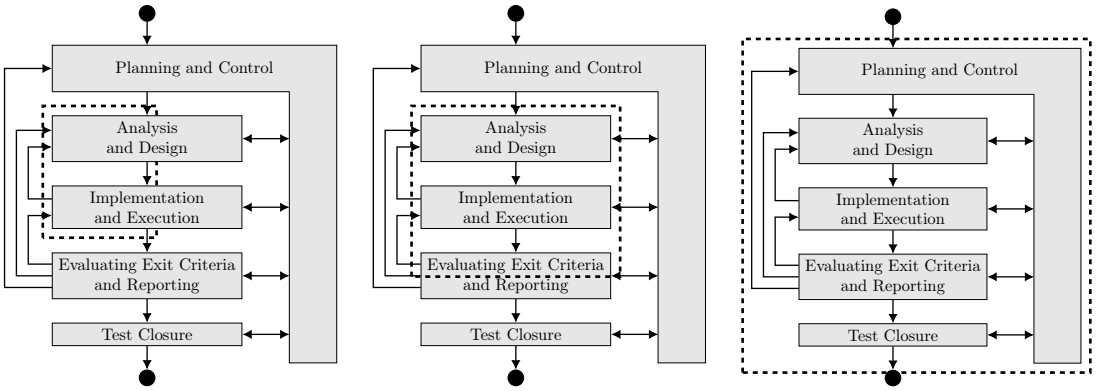
are applied in the area of software testing having its origins in work by Apfelbaum et al. [22]. In literature, there is a large number of definitions for Model-Based Testing (MBT), whereby these are usually formulated specifically for the respective application context. In some cases, MBT is only mentioned in the context of test case creation for black-box testing [168]. However, in other application areas, MBT is used for white-box approaches. Overall, MBT can be seen as a technique that derives the test cases wholly or partly from a model that includes certain aspects of the SUT [107]. Utting et al. split up the model-based testing process into five steps, explicitly arranging the new model artifacts in between the predominant STLC activities [168].

Since the underlying model plays a central role in MBT, the different types of models participating are discussed in more detail below. Among other authors, Winter et al. specify three different types of models, which are applied in the context of MBT [174]

- **Environment Model**
  This model maps parts of the context in which the SUD is embedded. For example, external influences on the system are modeled, which can concern boundary conditions, interfaces to other systems, or external factors.
- **System/Software Model**
  In contrast to the environment model, the system/software model represents the internal structure and behavior of the SUD. The structure describes the decomposition of the system into its components, which usually contain a subset of the functionality. The encapsulated low-level behavior together with interaction specifications emerges the overall behavior of the SUD on higher levels of integration.
- **Test Model**
  The Test Model includes information created and modified in test-related process steps. In particular, information on the test basis and test specification are modeled at this point. Therefore, different types of Test Models are available in the context of the MBT. On the one hand, existing models of the SUT, such as the system/software model, can be extended by the mentioned test-specific information or a Test Model can be derived. On the other hand, the Test Model can be a standalone model which reflects the tester's view of the requirements and may have a different level of abstraction compared to the previously mentioned model artifacts.

Depending on aspects like time, budget, the experience of developers/testers, and the project's context, the presented set of models is potentially reduced to one integrated model serving multiple purposes. Furthermore, it plays an important role in what purpose these models should serve. On the one hand, Test Models are used to make mental models explicit when testing. On the other hand, Test Models with clear syntax and semantics form the basis of an automated test toolchain, which, apart from the generation of test cases, may include their execution. Especially in the latter application scenario, some challenges have to be considered, to carry out a meaningful test process (see section 5.2.1).

Based on the different variants resulting from the role of models, a distinction is made between different forms of MBT as already shown in chapter 4 in the context of MDSD. Winter et al., therefore, define the terms *Model-Oriented Testing*, *Model-Driven Testing*, and *Model-Centric Testing* with the role of models becoming increasingly important across the variants [174] (see Figure 5.7).



(a) Model-Oriented Testing (b) Model-Driven Testing (c) Model-Centric Testing

Figure 5.7: Types of Model-Based Testing (subfigures based on [174])

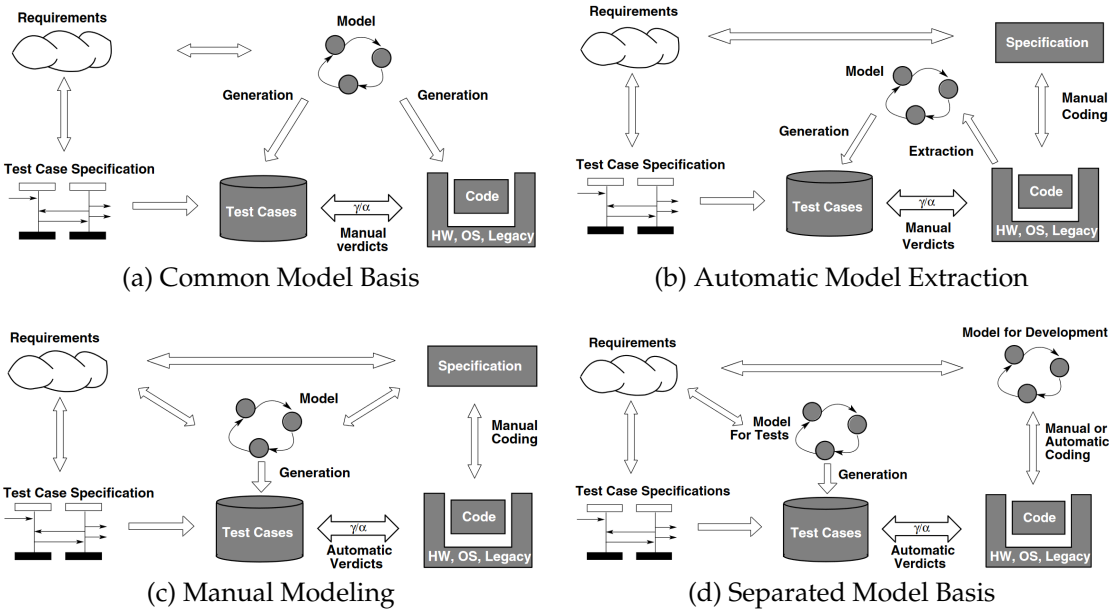
- **Model-Oriented Testing (figure 5.7a)**
  Model-Oriented Testing (MOT) represents the weakest type of MBT. Here, models are mainly used as a basis for expert discussions and manual reviews with a close relation to the basis of the original requirements. Due to the manual processing of model information and the main purpose being visualization, there is no need for detailed knowledge of the formal foundations. Concerning the classical STLC activities, only the phases *Analysis and Design* and *Implementation and Execution* are affected partially.
- **Model-Driven Testing (figure 5.7b)**
  In the context of Model-Driven Testing (MDT), modeling is assigned a more important role. In contrast to the MOT, various test artifacts are derived from models with suitable tool support. Depending on the level of detail of the model created, this step can be either fully automated or only partially automated. In the partially automated variant, skeletons are generated which are then completed by experts. Overall, the (partial) automation of recurring steps reduces the effort and the probability of errors. With this type of MBT, the activities *Analysis and Design*, *Implementation and Execution* and *Evaluating Exit Criteria* can be supported by the use of models, the *Reporting*, as well as other non-constructive activities, remain apart from the modeling approach.
- **Model-Centric Testing (figure 5.7c)**
  In contrast to the two variants of the MBT presented so far, the Model-Centric Testing (MCT) puts models into the center of testing. As illustrated in figure 5.7c, all activities of the STLC are affected by models. An important point is that models are not only a source of information but serve to represent the results of test activities. Furthermore, in MCT the level of abstraction or the change of the form of representation is rather avoided, which is why there is a significantly higher

overall transparency. This type of MBT is often used in combination with MDSD as a development paradigm, which leads to positive emergence effects.

All in all, there are many variants of how MBT can be realized. On the one side, the role of models in the individual test activities has a decisive influence. On the other side, it is crucial how the model basis is designed and for which purposes the modeled information can be used.

5.2.1 Scenarios of Model-Based Testing

Especially if MBT is to be implemented with a high degree of automation, there has to be an understanding of certain pitfalls regarding the models making up the information basis. This high degree of automation is usually reflected in the intention to derive both the application code and the test cases from a possibly multi-purpose model. Which constellations are possible is described in the following and is largely based on the statements of Pretschner et al.. [143]



(a) Common Model Basis (b) Automatic Model Extraction

(c) Manual Modeling (d) Separated Model Basis

Figure 5.8: Scenarios of Model-Based Testing (based on [143])

Common Model Basis

The first scenario is characterized by one single model for the two applications previously mentioned (see figure 5.8a). In particular, the set of test cases, as well as the target code, is automatically derived from this single model artifact. However, this setup

contradicts one of the basic concepts of testing, namely redundancy. The missing redundancy in the underlying model basis on the one hand requires manual test verdicts and on the other hand, limits the capabilities of thereby derived test cases to verify the adequacy of environment assumptions as well as the correctness of the applied code generator. However, this may not coincide with the actual intention of applying MBT concepts.

Automatic Model Extraction

The second scenario includes one single model artifact. While the previous scenario utilized the model in both contexts, this scenario just uses the model artifact to subsequently generate the set of test cases for the SUT (see figure 5.8b). The model itself is not derived from the basis of the requirements but rather extracted semi-automatic from the target code. In turn, the code is implemented manually based on a beforehand developed specification document. Identical to the previous case, the necessary redundancy is missing to be able to automatically derive test verdicts. Despite the indirection stage via the extraction of the model, no potentially diverse view of the requirements is created here. An obvious disadvantage of such a setup is the need for an extensive code basis to have the necessary information available for the extraction of a Test Model.

Manual Modeling

The logical next step of the previous scenario towards automatic test verdicts is given by manual modeling instead of deriving it from the code basis (see figure 5.8c). Thereby, not only the set of requirements is taken into consideration but the specification documents which serve as a basis for manual coding. This scenario is applied in distributed development setups or at least diverse teams for specification, implementation, and possibly testing. Further, hybrid forms of the *Automatic Model Extraction* scenario and the *Manual Modeling* scenario are often realized in a practical context.

Separated Model Basis

The last scenario detailed represents the optimal case in terms of the level of redundancy as well as the spirit of modeling (see figure 5.8d). Here, two separate model artifacts are derived from the requirements. One model is used for development purposes to improve the overall quality of either manually or automatically derived code. The other model just serves as a basis for ongoing test case generation. The two separate models further enable testers to use automatic test verdicts during test execution, which is a necessity for highly automated processes.

As we have seen, apart from the last approach, the processing of models included either in the constructive or the testing context, always needs manual support. Especially, in

continuous development setups, the number of applicable scenarios reduces to one, namely the one necessitating a separated model basis.

5.2.2 Model-Based Testing in Practice

When it comes to the practical application of MBT, the availability of appropriate tooling marks an essential point. Furthermore, adequate modeling languages are needed to make such technologies widely applicable and acceptable. Over the years, a very broad spectrum has been created in both areas.

From formal modeling languages to languages that emphasize visual presentation, there are a variety of solutions, often with application context-specific content. Specifically, certain model artifacts of the existing modeling scope of widely used languages, such as UML, are often used for MBT purposes, e.g. statecharts or activity charts for the specification of the SUT's behavior. [168]

At this point, another standard of the OMG, namely the UML Testing Profile (UTP), introduces concepts for carrying out MBT in the UML ecosystem. Technically, this standard is therefore realized as a UML profile, which covers large parts of the process steps known from the STLC. [4]

Zander et al. [178] and Schieferdecker et al. [154] therefore extracted the following groups of concepts from the mentioned standard.

- **Test architecture**
  This group covers the basic elements of a test and its relations. Essential parts are the so-called *test configurations* and *test components*, which create the basic setup for later test execution.
- **Test behavior**
  In this group, all elements regarding the dynamic aspects of test case execution are included. The most important concept is the *test case* itself, whose execution is specified by *test actions*, and the outcome is represented by *test verdicts*.
- **Test data**
  Concepts of this group cover the data handling and specification, which consequently lifts the set of abstract test cases to a set of concrete test cases. This functionality is implemented by model elements such as the *data pool*, *data specification*, *data partition* or *data items*.
- **Test planning**
  This group of model elements combines constructs of test analysis and test design. Essentially, this introduces structure-giving model elements that, for example, combine test cases into *test sets*, or the so-called *test context*, which logically combines multiple information in the context of the tests.

Altogether, this selection of modeling concepts covers almost all scenarios of MBT. The standard itself draws some possible usage scenarios, ranging from *the specification and*

design of test systems over *the creation of model-based test specifications for existing System Models* to *modeling test cases, test environments, and test data* [4].

Apart from the modeling language perspective, an excerpt of available tooling is presented throughout the rest of this section. First, a tool developed in an academic context by the TU Graz is mentioned, which combines model-based testing with mutation testing as presented in section 5.1.3. Here, either based on UML statemachines or Object Oriented Action Systems (OOASs), a set of test cases is derived using mutation operators. The tool called *MoMuT* represents a highly automated test case generation approach, which is still maintained and applied in a practical context. [115]

In contrast to the academic context, the commercial *Conformiq Creator & Designer* is briefly introduced. Conformiq supports various input formats. Possible modeling languages for processing range from UML activity charts and statemachines, to proprietary DSLs. The mentioned *Conformiq Creator & Designer* represent an excerpt of a tool suite enabling testers to further execute the specified test cases. Besides, many other formats of other tools in the context of MBT are supported, e.g. Testing and Test Control Notation (TTCN-3). [96]

The last tool presented here is called *mbtSuite* which is developed by AFRA GmbH, as well as sepp.med GmbH. In the context of the mentioned research project *ReTeC* [166] this tool was extended by functionality. The modeling basis is provided by UML activity charts and statemachines. Based on these, the test cases can be derived using different coverage criteria or even genetic or randomized algorithms. A connection to test management and execution tooling is also available. [156]

Part III

TOWARDS A MODEL-CENTRIC SOFTWARE TESTING LIFE CYCLE

6

General Approach and Running Example

Targeting the challenges emerging from a steadily rising level of development complexity, previously identified in section 1.1, we further draw a more concrete and holistic picture of our approach sketched in section 1.2. Finally, the running example, used throughout the main chapters of our work, is introduced.

To overcome these challenges, an MCSTLC is developed based on the process steps of a classic STLC (see section 5.1.1). The goals are the greatest possible automation of the process steps and the applicability in the early phases of development. This enables the use of an MCSTLC for quality control as well as a tool during development to consistently get hints for improvement.

6.1 General Approach

Throughout the rest of this chapter the phases of the MCSTLC are explained (illustrated in figure 6.1). These phases were originally sketched in one of our conference contributions [146]. The specified life cycle steps are related to the process steps of the original STLC, serving as the starting point. Besides, specific artifacts produced/consumed by the processing steps are addressed but not included in figure 6.1.

1. Model Creation/Modification marks the entry point of the MCSTLC and at the same time represents the interface to all disciplines in the development. The main tasks of this process step are given by the *Integrated Model Basis* (also called *Omni Model*) and the specification of configuration parameters like applicable coverage metrics, for the automated processing chain. Before executing the MCSTLC, all artifacts of the modeling domains involved in the development process have to be available and suitably integrated by the *Integration Model*. The Integration Model links information across the different development domain-specific model artifacts. Further, this Integration Model can be extended by adding content to align the different models (for details see chapter 7). This model artifact is the basis for all further process steps to obtain a fully automated iteration of the life cycle. However, at the end of each iteration of the MCSTLC manual adaptions are necessary.

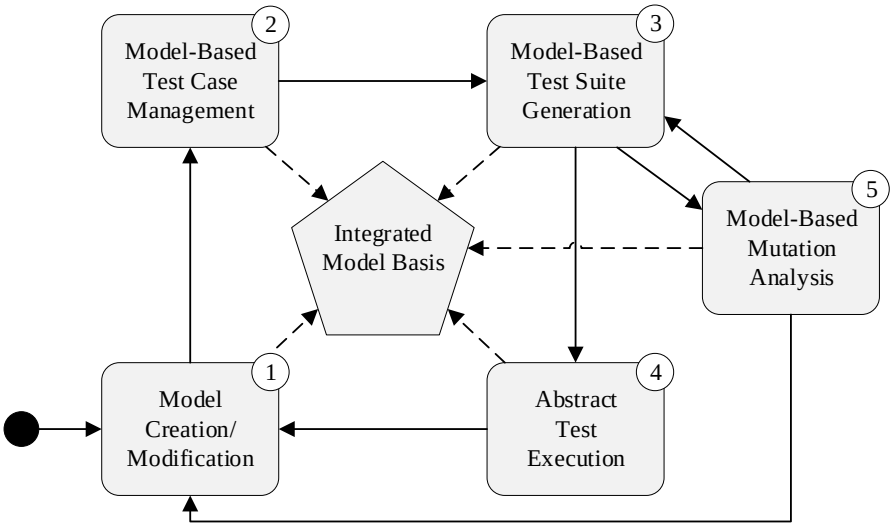


Figure 6.1: The Model-Centric Software Testing Life Cycle

From a tester's point of view, having the standard STLC in mind, activities from different process steps are combined in the context of the MCSTLC phase. The *Planning and Control* process step represents the counterpart, but there are also activities from *Analysis and Design*, as well as *Test Closure*, carried out in the current MCSTLC phase. In particular, the creation of a Test Model, representing a part of the Integrated Model Basis and forming the basis for the automated derivation of test cases are conducted in this phase. Further, measures either taken after completion of test activities or at the end of an MCSTLC iteration, are summarized under this phase.

Overall, the initial and the final phase of the life cycle represents the interface between humans and the automated processing chain. In particular, the information provided by the processing chain, namely the *Human-Interpretable Test Results* and the *Human-Interpretable Mutation Analysis Results*, should provide the user with sufficient support to a targeted test phase in the early stages of development. The elaboration in chapter 7 provides details and illustrates the application with a practical example.

2. Model-based Test Case Management represents the first phase of the automated processing chain. In this phase, different activities of the classic STLC are combined. Aspects of *Test Planning* and *Test Analysis* are realized by transforming the fully-blown Test Model into a strongly focused submodel. This is achieved by a multi-stage process, where model artifacts are pruned and projected to models of other development domains. All of the mentioned stages reside on the Integrated Model Basis or expert's configuration input. Moreover, aspects of *Test Design* and *Test Implementation* are applied at this phase. Due to the focussedness of the resulting Test Model excerpt, concepts like prioritization, selection, and reduction of test suites may be realized. Notice, that the concepts mentioned above are carried out before the complexity exposes in the context of the concrete set of derived test cases.

To realize this functionality, all available models are taken into account to derive a suitable portion of the original Test Model reflecting the tester's mindset. The carefully linked model data from various development domains of the Integrated Model Basis significantly impacts the quality of subsequent processing results. Details are presented in chapter 8 with an example.

3. Model-based Test Suite Generation takes as input the resulting *Scoped Test Model* and performs activities, which are encapsulated in the *Test Design* and the *Test Implementation* process steps of a classic STLC. Specifically, properties of the underlying models like contained data flow information, are evaluated and assessed together with configuration parameters, like exit criteria or a start metric, specified in the introductory phase of the life cycle. To determine an appropriate algorithm for the test case generation, a subsumption hierarchy of coverage metrics and respective implementations is utilized as a decision basis. The resulting test cases reflect the current test focus and represent the starting point for later iterations of the life cycle.

However, if there are test cases derived from this Test Model instance or similar Scoped Test Models in the form of *Machine-Interpretable Mutation Analysis Results*, an improved starting point for determining an appropriate test case extraction metric is given. The original set of test cases can be optimized and handed over to the next phase of the MCSTLC. The feedback from previous iterations enables the *evaluation of the exit criteria*, as in the classic STLC. Metrics on the model artifacts are evaluated after each iteration and provide a basis for termination of the cycle. Chapter 9 details and illustrates the application using an example.

4. Abstract Test Execution is the counterpart to the process activities *Test Execution* and partly *Test Reporting* of the standard STLC. This phase of the MCSTLC deals with the execution/interpretation of model artifacts from the integrated model basis. Therefore, the quality-proven set of test cases is analyzed about the integration level of test cases and the level of abstraction of the targeted System Model part. Based on the results of this analysis, two different kinds of abstract test case execution mechanisms are available. In case of model artifacts including no data flow information, a structural conformance check between the input test cases and the System Model linked via the Integrated Model Basis is performed. In the case of more concrete data embedded in the System Model, model interpretation based on derived paths through the SUT is carried out. Both approaches have in common, the valuable *Human-Interpretable Test Results* which further expand the expert's knowledge during the *Model Creation/Modification* phase. This process step enables testers to perform a test execution on model artifacts, whereby insights to the SUD at an early stage of development are gained. The elaboration in chapter 10 provides details and illustrates the application with a practical example.

5. Model-based Mutation Analysis represents a new process step not included in a classic STLC. The goal is to evaluate the quality of the created test cases concerning

a certain set of faults and thereby improve the resulting test suites. In contrast to the previous manual static checks, this is achieved by a flexible realization of mutation analysis on models. I.e. the test cases created in the *Model-based Test Suite Generation* phase are executed against targeted mutants of the existing System Model. Using the Integrated Model Basis the evaluation is carried out mainly automatically and is based on concepts that can be assigned to the *Test Execution* and *Test Reporting* of the STLC.

The results are used in other phases of the MCSTLC. This includes the *Model-based Test Suite Generation* phase, which can adjust the metrics for generating test cases based on *Machine-Interpretable Mutation Analysis Results*. However, the results can be made available to domain experts, i.e. *Human-Interpretable Mutation Analysis Results* are provided for the *Model Creation/Modification* step. Chapter 11 details and illustrates the application with a practical example.

6.2 Running Example: Ceiling Speed Monitor

In the section an example model is introduced, which is used to illustrate concepts in the following. The selected example should be sufficiently complex to demonstrate the practical applicability of the concepts developed as well as easy enough to understand the complexity of the problem.

The European Vital Computer (EVC) is a control unit for trains that conforms to the European Train Control System (ETCS) standard. The onboard controller, which controls the engines of the trains, includes a variety of functionalities. A dedicated group of functions of the controller deals with *Speed and Distance Monitoring*, reflecting the focus of our running example. The mentioned functionality is decomposed into three separate blocks, where each handles a certain subtask. An excerpt of the complete system which is focused on the context of the running example can be seen in figure 6.2.

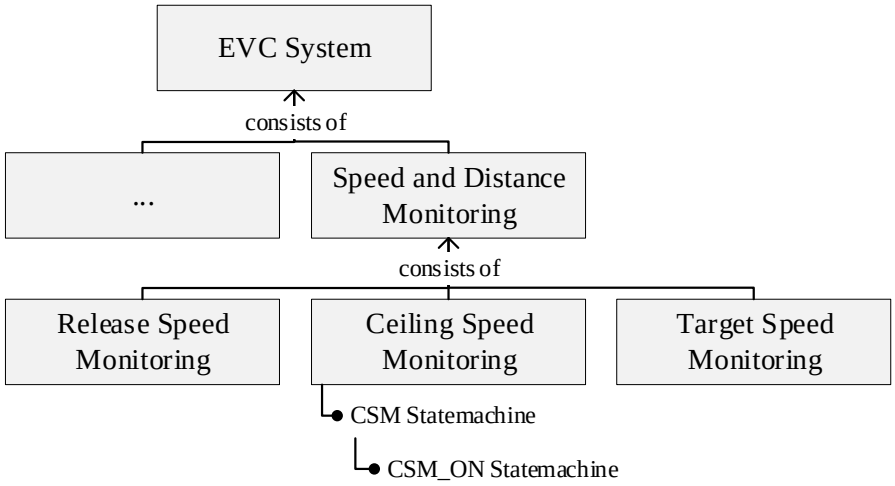


Figure 6.2: Excerpt of the ETCS EVC system

Braunstein et al. [38] define the functionalities as follows:

- **Ceiling speed monitoring (CSM)**
  supervises the observance of the maximal speed allowed according to the current most restrictive speed profile (MRSP). CSM is active while the train does not approach a target (train station, level crossing, or any other point that must be reached with predefined speed).
- **Target speed monitoring (TSM)**
  supervises the observance of the maximal distance-depending speed, while the train brakes to a target, that is, a location where a given predefined speed (zero or greater zero) must be met.
- **Release speed monitoring (RSM)**
  applies when the special target *End Of Movement Authority (EOA)* is approached, where the train must come to a stop. RSM supervises the observance of the distance-depending so-called release speed when the train approaches the EOA.

As part of the research work of the University Bremen in the context of MBT, an SYSML model for the CSM was created based on the requirements of the ETCS system. The requirements concerning the CSM system are contained in the model and are further referenced in the context of the main chapters. The actual model is divided into different submodels. The top-level structures the model into the actual SUT and the test environment, as well as the intended interfaces and included data structures, which can be seen in figure 16.1 (Supplementary Material). Furthermore, the SUT splits up into the previously mentioned subsystems, CSM, TSM, and RSM. At this point, the original model of the University Bremen was extended by the blocks TSM and RSM. Altogether, these blocks are embedded inside the SUT block (see figure 16.2 (Supplementary Material)) and are used to demonstrate some features of the approach.

The SYSML block `CSM` contains the specification of behavior as a statemachine (see figure 16.3 (Supplementary Material) and figure 16.4 (Supplementary Material)). The statemachine is hierarchically structured and reflects the state of the CSM subsystem. A change between the states is caused by the change of external influences and managed by method calls, which are referenced by the CSM system. These methods represent the smallest functional units of the system under consideration.

The detailed model artifacts of the System Model are presented in part VI (Supplementary Material). Additional model artifacts like behavioral models or Test Models including the respective metamodel are defined in chapter 7.

7

Omni Model Approach

As already described in the introduction of the MCSTLC, a meaningful data/model basis plays an essential role in MDSD. In particular, large parts of the development and the validation process should be automated. The process step of creating and modifying model artifacts is the entry point into our model-centric instance of an STLC as well as the point of interaction between experts and the automated MCSTLC chain. Figure 7.1 shows a portion of the entire MCSTLC including the used model artifacts and their exchange between the different process steps.

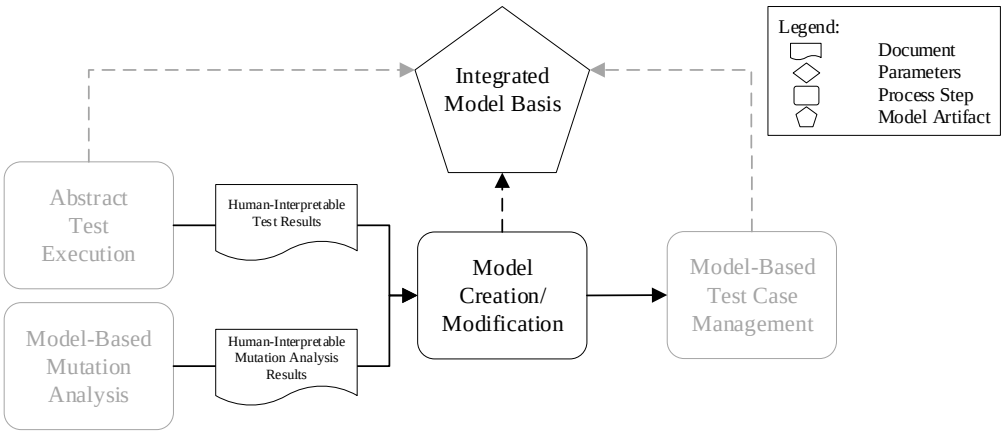


Figure 7.1: MCSTLC extract focusing on the creation and modification of the Omni Model and involved information

The process step *Model Creation/Modification* consumes different types of feedback from previous iterations of the MCSTLC to correct defects or possible improvements. Moreover, this process step defines the necessary parameters for the configuration of the following processing chain and optimizes them based on the feedback.

The most important component is the central knowledge basis, which is created or manipulated by the process steps of the MCSTLC and acts as the model data sink. This Integrated Model Basis, called Omni Model, represents an integration of model information from implicitly interconnected development domains like MDSD, MBT, and Model-Driven Multi Concerns Analyses. By integrating the separated models, knowledge about the developed system is bundled and made available to all disciplines of development. In the context of the MDA methodology, the Omni Model approach is

primarily on the *Platform Independent Models (PIM)* level (shown in figure 7.2). In later phases of the MDSD, variants of the target code artifacts can already be derived, whereupon the execution of the steps of the STLC leads to more concrete results. Nevertheless, the Omni Model approach can be applied in these phases for improvements, but are not the focus of this thesis.

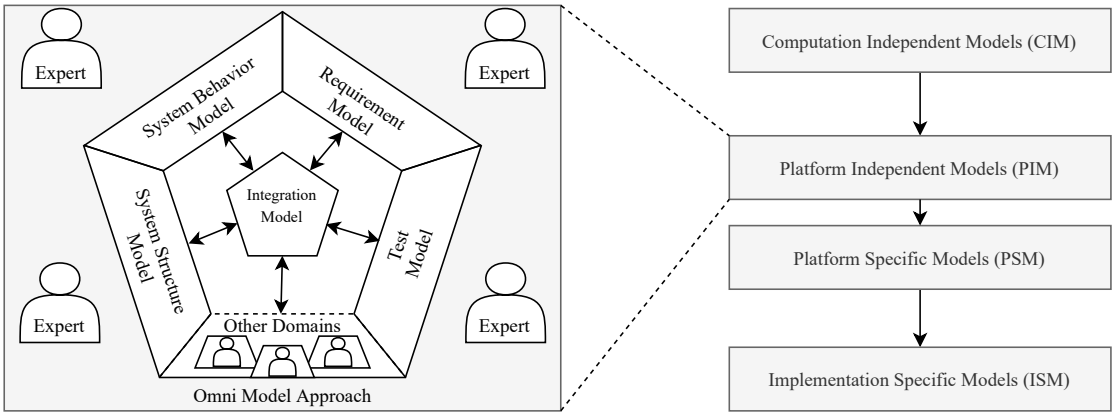


Figure 7.2: Integrated Model Basis in the context of MDA

Apart from the classification into the phases of MDA, figure 7.2 illustrates the concept of integrating the model artifacts. In particular, an expert approach is to be implemented, i.e. in the respective domains, experts are responsible for the choice of the modeling language and the subsequent modeling of the relevant aspects. The choice of the respective modeling language remains unaffected by the Omni Model approach as long as the underlying metamodel and the associated syntax and semantics are known. Therefore, the experts work in their accustomed environment with their experience. Thus, this design decision keeps the complexity of the model basis as low as possible. Besides, a mapping of concepts across the domains is established and maintained for structural as well as behavioral models of SUD. This allows information to be used across domains in a controlled manner.

7.1 Domain-Specific Models

The Integrated Model Basis represents a flexible composition of domain-specific models. In this section, the minimum set of domains, necessary to realize all components of the presented MCSTLC, is examined in detail. This includes the *System Modeling Domain*, the *Test Modeling Domain*, and the *Integration Modeling Domain*. From these domains, different modeling languages are discussed, which are either used in the context of the running example or the evaluation examples and are therefore relevant for further understanding. The modeling languages discussed in more detail, i.e. metamodel concepts are explicitly tackled and set into relation, represent the set of modeling languages applied in the running example. The other languages are further applied during the case studies, while details for these metamodels are shown in part VI (Supplemen-

tary Material). In addition to DSMLs in the classical sense, GPMLs are briefly discussed and how they can be used in the respective domain.

7.1.1 System Structure and Behavior Metamodels

First, we focus on the software modeling domain affecting the resulting artifacts of the constructive phases of development. As usual, we distinguish between structure- and behavior-describing models. Depending on the use case and development context, there is a wide range of possible modeling languages that can be applied. The spectrum ranges from very formal modeling languages to languages that are primarily used for visualization purposes. In the context of the presented MCSTLC, the modeling languages need concrete semantics to provide a sufficient basis for subsequent processing steps.

In the following, some modeling languages and the underlying metamodels are described in more detail. The languages presented are applied in the context of the running example and the case studies considered in the evaluation chapter. The former is described in more detail and set about concrete model instances of the running example.

OMG Systems Modeling Language

The Systems Modeling Language (SYSML) of the OMG is a popular language in the field of systems engineering. The SYSML is based on UML and enables the structure and behavior of the respective SUD to be modeled by many different diagram types [1]. For the sake of clarity and demonstration purposes of the approach presented in this thesis, a simplified metamodel of SYSML is defined based on the SYSML model of the *Ceiling Speed Monitor* (introduced in section 6.2). The metamodel is specified as an Ecore model since this technology offers the best integration within the prototypical implementation of the concept (see section 7.3). A complete mapping of the concepts covered in the SYSML specification is possible but is not intended in the context of this thesis.

In the following, the simplified metamodel of SYSML is introduced and the corresponding SYSML model of the running example is explained and set into relation with the metamodel. We start with the top-level concepts and packages, which form the basis for all further specific concepts (see figure 7.3). This excerpt shows the most general concept of the metamodel, identifying any model artifact. Furthermore, concepts are introduced which allow to specify attributes and parameters.

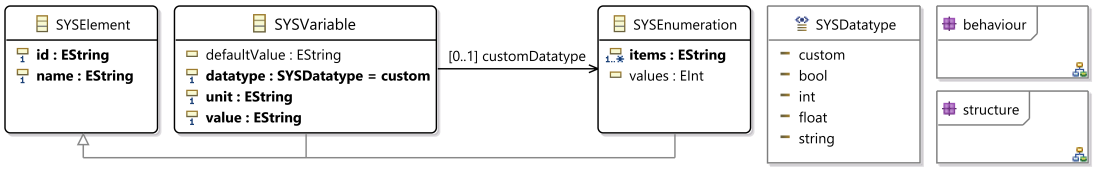


Figure 7.3: Simplified SYSML metamodel

Table 7.1: Metamodel element descriptions for figure 7.3

Concept	Description
`SYSElement`	The most abstract concept of the simplified SYSML metamodel with a unique `id` and `name` attribute
`SYSVariable`	Concept for the definition of block member variables
`SYSEnumeration`	Concept for the representation of enumeration datatypes
`SYSDatatype`	Enumeration for the supported basic datatypes of a block member variable

Below this top-level of the SYSML Ecore metamodel, two additional packages are included: First, the *structure* package (figure 7.4 and table 7.2) defining concepts to structure the SUD by blocks, to define their interfaces, and to map the information flows between these structural elements.

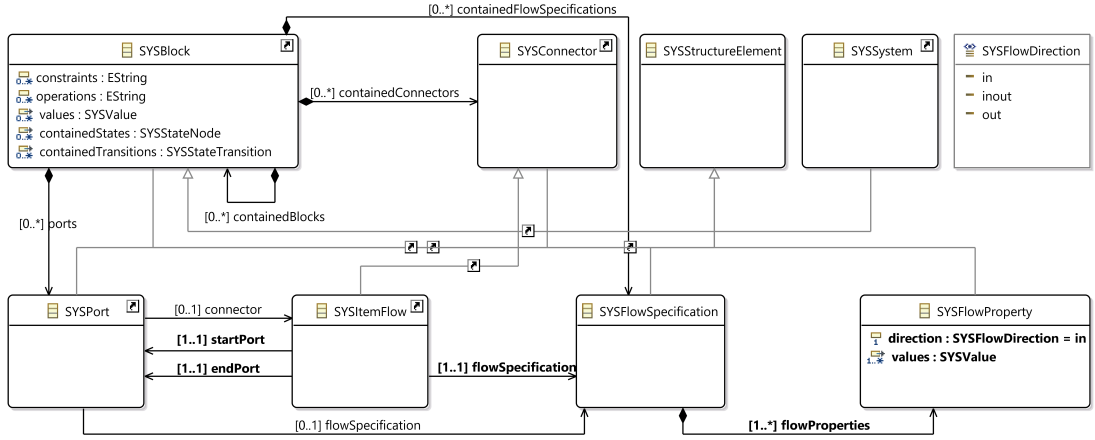


Figure 7.4: `Structure` package of the simplified SYSML metamodel

Table 7.2: Metamodel element descriptions for figure 7.4

Concept	Description
`SYSStructureElement`	The most abstract concept of the structure package (for implementation purposes)
`SYSBlock`	Concept reflecting the standard SYSML block
`SYSSystem`	A special kind of SYSML block encapsulating the overall system
`SYSPort`	Concept for the SYSML port which is attached to a block
`SYSConnector`	Abstract concept representing multiple types of connectors in the SYSML context
`SYSFlowDirection`	Enumeration for the specification of an information flow direction regarding the mentioned port concept
`SYSItemFlow`	Concept contributing the specification of an item flow between two ports
`SYSFlowSpecification`	Concept encapsulating a specification for the items exchanged through an item flow
`SYSFlowProperty`	Concept encapsulating the valid direction and data structures for the respective ports

The second package, namely the *behavior* package, is shown in figure 7.5 and table 7.3. Not all behavior diagrams specified by UML and SYSML are shown here. The concepts defined in figure 7.5 allow the specification of hierarchical statemachines.

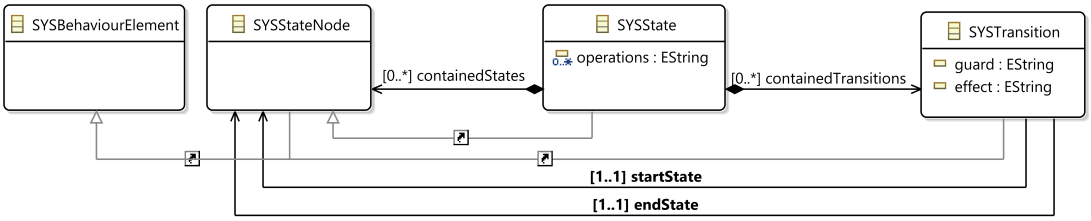


Figure 7.5: `Behavior` package of the simplified SYSML metamodel

Table 7.3: Metamodel element descriptions for Figure 7.5

Concept	Description
`SYSBehaviourElement`	The most abstract concept of the behavior package (for implementation purposes)
`SYSStateNode`	Concept for a state of the statemachine not specifying any behavioral details, e.g. initial or final nodes
`SYSState`	Concept for a state including additional information like triggered operations

Table 7.3: Metamodel element descriptions for figure 7.5 (continued)

`SYSTransition`	Concept for a transition between two state nodes with `guard` conditions and triggered `effects`

With this set of concepts, it is possible to completely model the *Ceiling Speed Monitor*. Thereby, models based on this metamodel can be used in the context of the Omni Model approach and thus applied in later process steps of the MCSTLC. Part VI (Supplementary Material) shows the individual diagrams describing the entirety of the *Ceiling Speed Monitor*.

OMG Unified Modeling Language

An alternative modeling language for the system model parts of the Omni Model is UML. This GPML, which covers many disciplines of development, offers a very wide range of concepts and diagram types for MDSD [3]. The modeling scope of UML is significantly larger than that of the SYSML, which is reflected in a high number of diagram types for structural and behavioral modeling. However, a complete mapping would again exceed the scope of the work. Therefore, the metamodel used for UML represents a simplified variant created about the model artifacts of the case study (see section 12.2).

The metamodel is divided into several packages that group the concepts according to diagram types as well as responsibilities. Part VI (Supplementary Material) introduces and explains all concepts of the simplified metamodel and puts them into relation.

The radCase Modeling Approach

The last modeling language, which is used exemplary for the System Modeling domain is given by the modeling tool *radCase* of the company IMACS [75]. This language is used to model embedded systems and therefore offers specific concepts for this area. Besides, the modeling language is implemented as a UML profile, i.e. any GPML modeling tool can be used to create radCase models. Due to the strong focus on the application domain, the underlying metamodel is not very elaborated. This metamodel is the basis of the System Models in sections 12.1 and 12.3. In part VI (Supplementary Material) this metamodel is considered more closely.

(Modeling) Guidelines & Best Practices

A set of guidelines and best practices are necessary to ensure the basic functioning of subsequent processing steps as well as the quality of resulting findings. It is important

to note that the overall structure in System Modeling has a significant influence on the subsequent modeling of the test cases. If a certain amount of preliminary work is already being carried out on the System Modeling side, then the Test Modeling can be carried out effectively. However, if important guidelines are disregarded, a system can hardly be tested resulting in remaining serious defects. Therefore, this section defines metamodel-independent modeling guidelines useful for the MCSTLC steps.

The testability of a system is affected by its transparency regarding connectors between components and the structuring of sub-components. These properties have proven to be particularly useful during validation of hardware-related components under the term "Design4Testability" [173]. In the context of System Modeling, these properties can be achieved by a wide variety of techniques. For instance, contract-based as well as component-based approaches represent techniques favoring the mentioned transparency and therefore have positive effects on the testability of the system.

Contract-Based Design reduces the complexity of a SUD by clearly exposing interfaces of (sub-)systems and defining their interaction in form of contracts avoiding hidden links and emergent effects. Sangiovanni-Vincentelli et al. refer to a horizontal reduction of complexity, whereas the structuring of the model represents a vertical reduction of complexity [153]. In the context of our Omni Model approach, the Integration Model explicitly defines a property element to specify contract information for the respective System Model parts.

Component-Based Design and Patterns have positive effects on the testability of the SUD, since especially the integration levels are explicitly expressed by the model artifacts. Having a possible state space explosion in mind, Groote et al. have formulated some guidelines with positive effects on the testability of the system [81]. In particular, guidelines `III`, `VI`, and `VII` explicitly address transparency concerning external behavior, the combination of components, and their interaction.

A mapping of classical design patterns from object-oriented software development promises a positive effect in the context of MDSD. For example, a strict application of structural patterns in the sense of the *Facade* pattern results in highly structured models, which in turn enables fine-grained integration levels. Behavioral patterns such as the *Strategy* pattern further promote the definition of explicit interfaces of functional components and thus improve their testability. [72]

7.1.2 Test Metamodels

MBT represents a model-based methodology in the context of testing. Furthermore, an essential aspect of the Omni Model approach is that each development domain, including testing, maintains its models. A wide variety of modeling languages exist. The selection of languages is based on the modeling languages used in the context of the

running example and the case studies. The modeling languages applied in the running example are discussed in more detail and visualized by respective model instances.

OMG UML Testing Profile

The UML Testing Profile (UTP) is a UML profile developed for modeling artifacts related to testing. The language scope of the UTP covers the areas of *Test Architecture*, *Test Behavior*, *Test Data*, and *Test Planning*. Thus, this modeling language allows to map test cases and to model other artifacts that are part of the STLC. Again a simplified metamodel was created, which is oriented to the modeling scope of the model instances based on it. This is particularly relevant in the context of the Ceiling Speed Monitor Test Model discussed after the presentation of the metamodel concepts. Figure 7.6 shows the most basic concept of the metamodel and the packages it contains.

Figure 7.6: Simplified UTP metamodel

Table 7.4: Metamodel element descriptions for figure 7.6

Concept	Description
`U2TPElement`	Abstract concept for a uniform basis
→`name`	String attribute for the specification of a `name`
→`id`	String attribute to specify a unique identifier for each element

Other concepts of the UTP are divided into three packages: The *architecture* package (figure 7.7, table 7.5) defines the `U2TPTestContext`, which sets the `U2TPTestCase` including the `U2TPTestData` and the respective `U2TPTestComponents` with each other. Furthermore, the included components can reference a different context, which represents a deeper level of the overall Test Model.

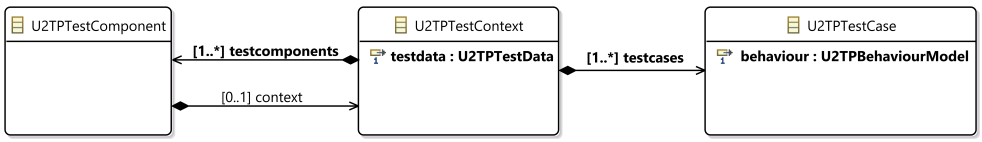


Figure 7.7: `Architecture` package of the simplified UTP metamodel

Table 7.5: Metamodel element descriptions for figure 7.7

Concept	Description
`U2TPTestComponent`	Concept for a component participating in a test case
`U2TPTestContext`	Concept integrates all the information needed for carrying out testing
→`testdata`	`U2TPTestData` attribute filling abstract test cases with concrete data
`U2TPTestCase`	Concept for a abstract test case
→`behaviour`	`U2TPBehaviourModel` attribute to determine the behavioral description of a test case

Modeling the behavior of a test case plays a central role. Therefore, the *behavior* package (figure 7.8, table 7.6) provides a simplified metamodel, which is limited to sequence diagrams consisting of different `U2TPLifelines`, which correspond to the `U2TPTestComponents` specified in the respective `U2TPTestContext`. `U2TPMessages` can be sent between the lifelines, which in turn can be encapsulated in `U2TPFrames` to model more complex behavior.

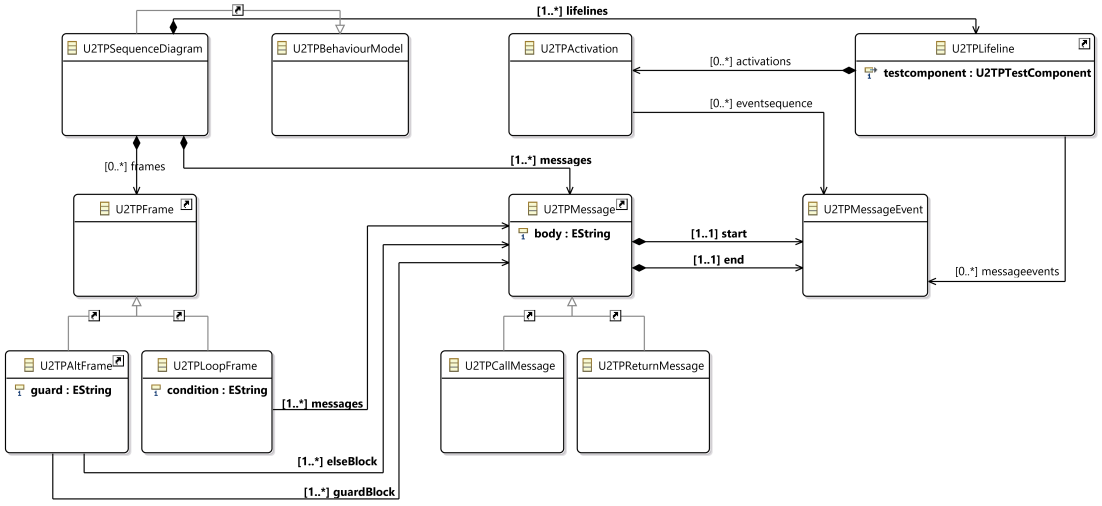


Figure 7.8: `Behavior` package of the simplified UTP metamodel

Table 7.6: Metamodel element descriptions for figure 7.8

Concept	Description
`U2TPBehaviourModel`	Abstract concept for various types of models for the specification of test behavior
`U2TPSequenceDiagram`	Concept for the specification of test behavior as a UML sequence diagram

Table 7.6: Metamodel element descriptions for figure 7.8 (continued)

U2TPActivation	Concept for the activation event of a U2TPLifeline of a U2TPSequenceDiagram
U2TPLifeline →testcomponent	Concept for a lifeline of a U2TPSequenceDiagram U2TPTestComponent attribute for the specification of an affected component
U2TPFrame	Abstract concept for the definition of special constructs in the context of U2TPSequenceDiagrams
U2TPAltFrame →guard	Concept for an alternative flow in a U2TPSequenceDiagram String attribute for the specification of a boolean guard expression
U2TPLoopFrame →condition	Concept for a loop in a U2TPSequenceDiagram String attribute for the specification of a condition expression
U2TPMessage →body	Abstract concept for the information exchange or method call between two U2TPLifelines of a U2TPSequenceDiagram String attribute for the payload of a message
U2TPCallMessage	Concept representing a call message from one lifeline to another
U2TPReturnMessage	Concept representing the return value of a previously called method
U2TPMessageEvent	Concept to determine the start and end of U2TPMessages on the respective U2TPActivations of lifelines

To transfer the behavior specified in the form of a sequence diagram into concrete tests, test data must be modeled. This is achieved by the concepts of the *data* package (figure 7.9, table 7.7). All data is managed in the U2TPDataPool, divided into different U2TPDataPartitions. This defines how the data structures look and which concrete instances (U2TPDataPartitionInstances) are specified for the intended set of concrete test cases.

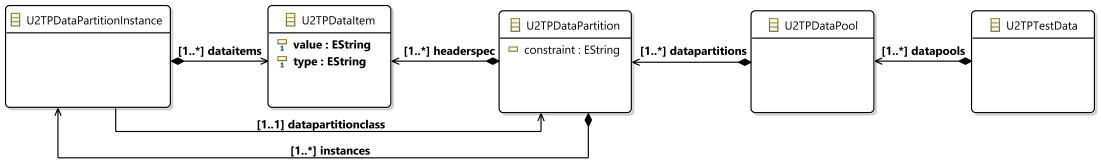


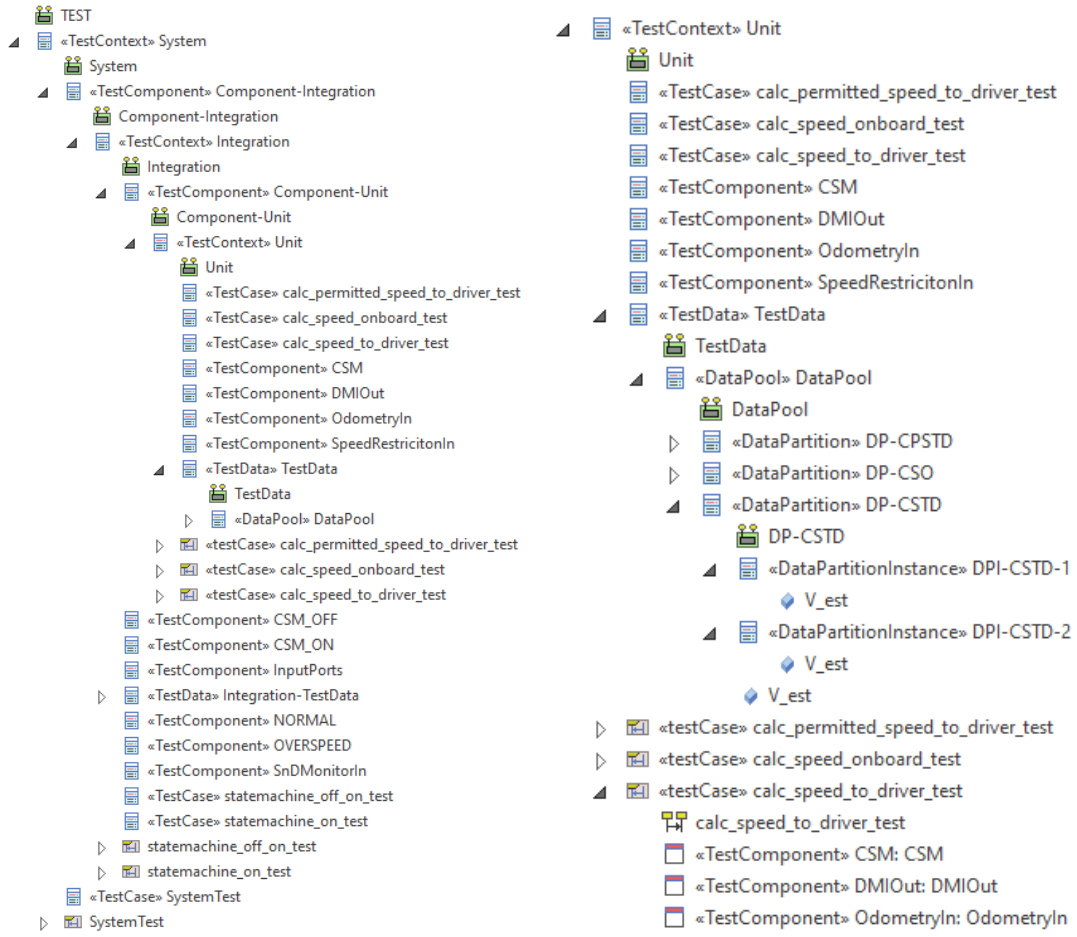
Figure 7.9: Data package of the simplified UTP metamodel

Table 7.7: Metamodel element descriptions for figure 7.9

Concept	Description
U2TPDataPartitionInstance	Concept for the specification of an instance of the defined U2TPDataPartition enabling the derivation of concrete test cases from a U2TPBehaviourModel
U2TPDataItem	Concept for the specification of attributes making up a U2TPDataPartition
→value	String attribute for the specification of a certain data value
→type	String attribute for the specification of a data type for the respective attribute
U2TPDataPartition	Concept structuring a U2TPDataPool regarding certain sets of abstract test cases
→constraint	String attribute for the specification of constraints for the U2TPDataItems making up the data partition
U2TPDataPool	Concept for the organization of test data

Altogether, the presented constructs allow specifying a Test Model in UTP, which participates in the Omni Model. In the following, excerpts of the UTP Test Model of the Ceiling Speed Monitor are introduced, since this forms the basis for the demonstration of subsequent processing steps.

Figure 7.10a shows the basic breakdown of the Test Model into different levels of integration. This decomposition is based on the subdivision of the SUT into subsystems and the available interfaces, which significantly influence the number of integration levels. Per test level, some U2TPTestContexts encapsulate all information for a set of Test Cases. The test behavior models are implemented as sequence diagrams, combined with the test data from the U2TPDataPools during the generation of concrete test cases. Figure 7.10b shows different unit tests. Especially the U2TPDataPartition DP-CSTD specifies two different instances (DPI-CSTD-1 and DPI-CSTD-2) for the test behavior model calc_speed_to_driver_test.



(a) General structure (b) Unit level testmodel

Figure 7.10: Excerpts of the UTP Test Model of the CSM system

The mbtSuite Modeling Approach

There are a lot of possibilities for modeling test cases. Besides the UTP, based on the UML, there are commercial tools based on proprietary modeling languages, e.g. the *mbtSuite* of sepp.med GmbH [156]. The *mbtSuite* allows the creation of Test Models in an extended form of UML activity diagrams, as well as a variant of the UML statecharts. Furthermore, concrete test cases can be derived based on such models and exported into concrete test scripts of different programming languages.

Concerning the case studies, a simplified metamodel is created. This metamodel only considers the activity chart part and is described in more detail in part VI. Such a model consists of nodes and edges, where the nodes describe the possible steps in a set of Test Cases. The nodes are divided into *Test Steps* (nodes that send stimuli to the SUT) and *Verification Points* (nodes that check the system state). The edges in this graph define in which order the steps can be combined and which paths (Test Cases) can be derived. Especially loops, conditional test sections, or even initialization phases can

be constructed. Furthermore, there are structured nodes, which can include other Test Models. Thereby, hierarchies in Test Models can be modeled, which corresponds to the integration levels in the test context.

(Modeling) Guidelines & Best Practices

As stated in the context of System Modeling, some guidelines can be formulated for Test Modeling as well as to improve the knowledge and artifacts of the subsequent process steps. These guidelines apply to all concrete instances of modeling languages used in the test domain and thus are formulated in general terms. Starting from a solid test design, which is based on the established techniques described in the foundations (see section 5.1.3) additional guidelines may improve the overall quality. Therefore the following guidelines should be considered:

1. The Test Modeling should have a structure aligned with the decomposition of the SUT into its subsystems and functional units.
2. The Test Behavior Models should represent concrete initialization phases such that the expected system state can be guaranteed before the actual test case starts. This is particularly important when test cases are executed against the same instance of the system.
3. The Test Models of the different test levels should be strongly focused on to better identify possible error causes. Ideally, each test behavior model should examine one aspect of the SUT.
4. The ratio of test steps and verification points should be balanced to be able to draw clear conclusions about error causes. This is especially valid in the white-box context, which provides insights into System Modeling apart from the interfaces.

The last two aspects lead to better results, especially when evaluating the resulting test cases (see chapter 10). However, beyond the model-level, this property of the test design cannot always be fulfilled, because this information of the system is not available at this point.

7.1.3 Integration Metamodel

The Omni Model approach is a combination of model artifacts from different development domains. There is no upper limit to the number of contributing domains, but at least the System Modeling domain and the Test Modeling domain have to be present. The artifacts of the development domains are specified according to different metamodels which do not have any context information about other development domains. To keep responsibilities and information cleanly separated, the Omni Model approach introduces the so-called *Integration Model*. It represents an additional model, which partly requires expert knowledge and parts can be derived/generated from other models of

the Omni Model. The Integration Model allows to explicitly link structural and behavioral model parts. Parts of the system, which cannot be assigned to a specific domain, can be captured as well. These functionalities allow the Omni Model to be used in its entirety as a valuable basis for the MCSTLC.

Like all other models of the Omni Model, the Integration Model is defined by a metamodel. Note, abbreviations are further on used for a more compact presentation. The running example is used to illustrate the introduced concepts.

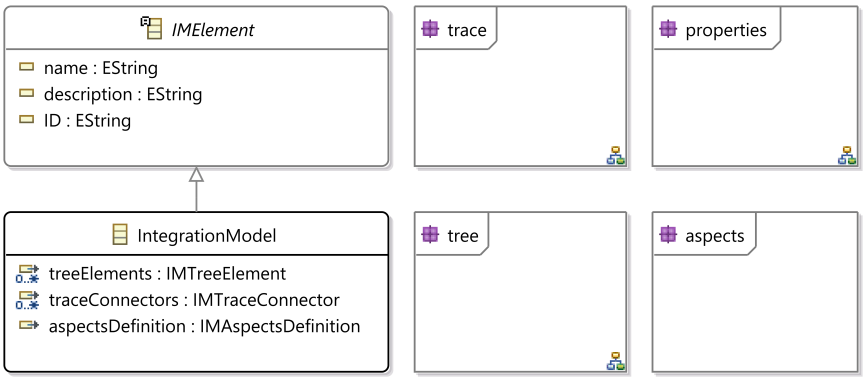


Figure 7.11: Integration Model (IM) metamodel

Table 7.8: Metamodel element descriptions for figure 7.11

Concept (Abbreviation)	Description
`IMElement (im_el)`	Abstract concept for all the included metamodel concepts of the IM
→`name`	String attribute which represents the name of the element
→`description`	String attribute which captures the description of the element
→`ID`	String attribute which reflects the unique identifier of the element
`IntegrationModel (im_im)`	Concept which acts as a container for all the model elements included in an instance of the IM
→`treeElements`	Set of `IMTreeElements` making up the tree structure of the Integration Model
→`traceConnectors`	Set of `IMTraceConnectors` specifying the mapping information included in the Integration Model
→`aspectsDefinition`	`IMAspectDefinition` for the IM (detailed in the context of the Aspect Definition Language)

As can be seen in figure 7.11 and table 7.8, an abstract concept (`IMElement`) is defined making each object unique by an `ID`, meaningful `name`, and an optional `description`. Furthermore, a model element `IntegrationModel` is introduced which acts as

a container encapsulating model elements, e.g. the `IMAspectDefinition` (see section 7.1.3). The inheritance between concepts, `IMElement` and `IntegrationModel`, is shown, whereby the attributes are assigned to the child concept. In addition to concrete concepts of the metamodel, figure 7.11 shows the packages, containing the more advanced concepts on which the structure of the following sections is based.

Structural Concepts

The structural components of the Integration Model are defined in the `tree` package. An instance of the Integration Model reflects a tree structure that shows the decomposition of the instantiated SUT into its functional components and their encapsulation. In particular, this allows the simplest and most intuitive representation of the SUT to be chosen as a basis for discussion between experts of the development domains. To enable this representation, which is created either automatically from instance diagrams of the System Modeling domain or manually, the following concepts are provided in the metamodel (see figure 7.12, table 7.9).

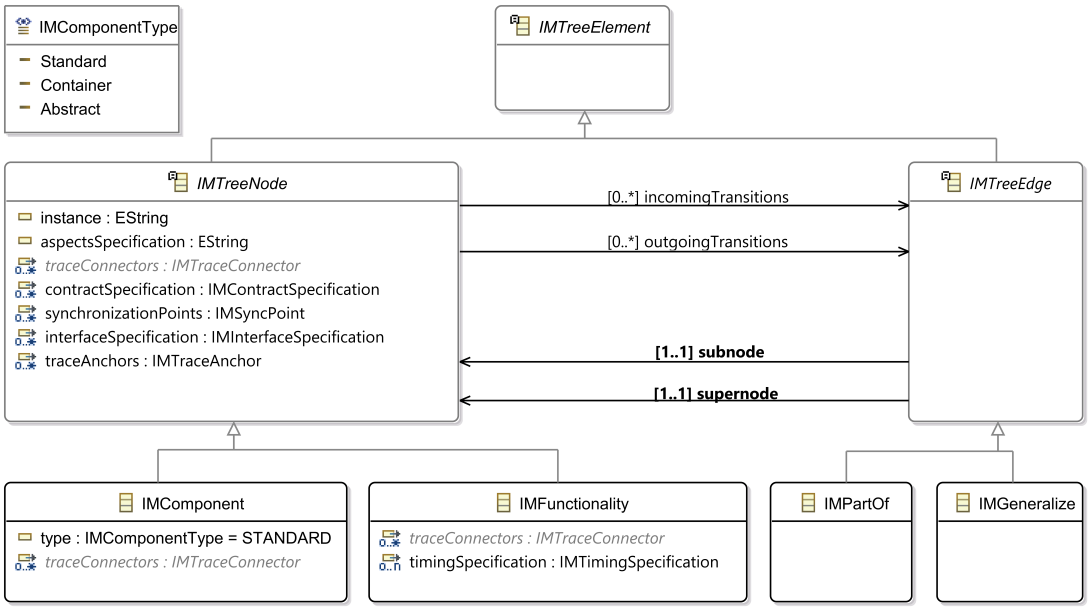


Figure 7.12: Tree package of the IM metamodel

Table 7.9: Metamodel element descriptions for figure 7.12

Concept (Abbreviation)	Description
`IMTreeElement (im_te)`	Abstract concept for implementation purposes of the tree package
`IMTreeNode (im_tn)`	Abstract Concept representing a node of the tree structure reflecting the breakdown of the SUT
→`instance`	String attribute capturing the unique instance identifier

Table 7.9: Metamodel element descriptions for figure 7.12 (continued)

→aspectsSpecification	String attribute capturing the aspect specification which holds for the node of the tree structure
→traceConnectors	IMTraceConnectors which specify a semantic relation to model elements of other development domains
→contractSpecification	IMContractSpecification to specify contracts which hold for the related elements of the SUT
→synchronizationPoint	IMSyncPoint for the detailed specification of mappings between model elements of different development domains
→interfaceSpecification	IMInterfaceSpecification to determine interface information to synchronize interface information
→traceAnchors	IMTraceAnchors to specify detailed semantic relations to model elements of other development domains
IMComponent (im_co)	Special concept of the tree structure representing structure-defining elements of the SUT breakdown
→type	IMComponentType attribute which defaults to the Standard value
→traceConnectors	IMTraceConnectors which specify a semantic relation to model elements of other development domains
IMFunctionality (im_fu)	Special concept of the tree structure representing behavior-defining elements of the SUT breakdown
→traceConnectors	IMTraceConnectors which specify a semantic relation to model elements of other development domains
→timingSpecification	IMTimingSpecification specifies abstract timing information between IMSyncPoints
IMComponentType (im_ct)	Enumeration concept to determine the type of a IMComponent: Standard, Container, Abstract
IMTreeEdge (im_tt)	Abstract concept for a connector of the tree structure
IMPartOf (im_po)	Concept reflecting the inclusion of the tree node at the lower level of this relation
IMGeneralization (im_ge)	Concept reflecting an inheritance relation between the connected nodes

The so-called IMTreeNodes represent the intermediate node and leaves of the tree structure, which are connected by IMTreeEdges. The nodes are further distinguished between IMComponents, which represent the structural components of the system, and IMFunctionalities, which address the behavioral descriptions of the System Model. Both types of nodes provide the ability to specify certain detailed information (see section 7.1.3). At this point, only the IMComponentType is referred to, which allows a distinction between different types of structural components of the system. There are two types of transitions, the IMPartOf relation, which indicates that the child elements are included, and the IMGeneralization, which is used for generalization.

A concrete instance in the context of the running example is given. Figure 7.13 shows the tree structure, which is reflected by the Integration Model instance (gray rectangles).

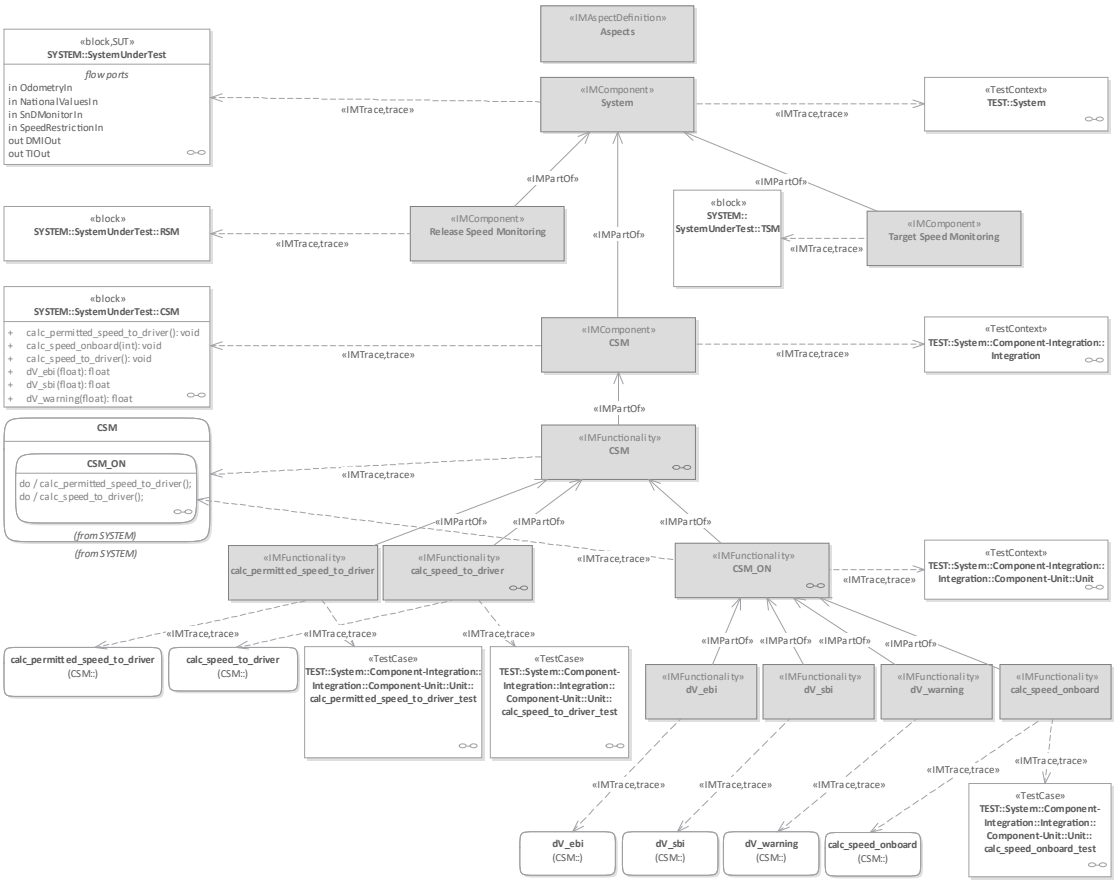


Figure 7.13: Integration Model structure for the CSM system

In this example, there are no major differences between the system's structure and the structure of the Integration Model because there are no multiple instances of the System Model parts. The same holds for the relation between the Test Model and the Integration Model.

Mapping Concepts

It was already stated that the Integration Model manages the combination of different domain-specific model artifacts and especially the included metamodels. That is, the integration is achieved by the constructs contained in the *trace* package. Figure 7.14 gives an overview of the concepts of this package, while short descriptions are provided in table 7.10. Overall, the concepts of the metamodel presented in the following provide the basis for the controlled use of information beyond the boundaries of domain-specific models. This enables all further processing steps within MCSTLC.

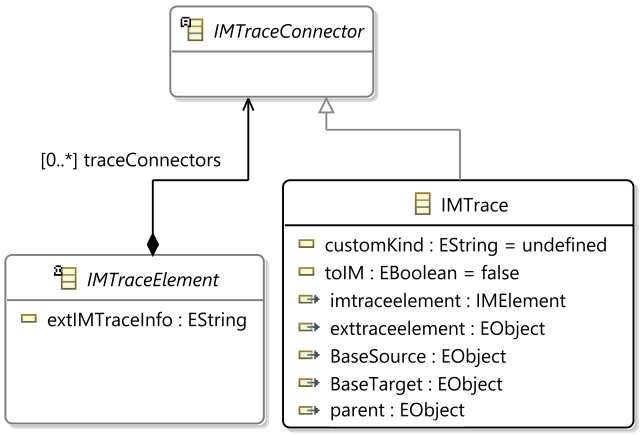


Figure 7.14: Trace package of the IM metamodel

Table 7.10: Metamodel element descriptions for figure 7.14

Concept (Abbreviation)	Description
`IMTraceConnector (im_trc)`	Abstract concept for implementation purposes of the trace package
`IMTraceElement (im_tre)`	Concept enabling another concept of the metamodel to specify mapping relations
`→extIMTraceInfo`	String attribute to specify external mapping information
`IMTrace (im_tr)`	Concept for the specification of mapping information between model elements
`→customKind`	String attribute to determine the development domain of the connected element
`→toIM`	String attribute to determine the direction of the mapping relation
`→imTraceElement`	`IMElement` which reflects the Integration Model part of the focused mapping relation
`→extTraceElement`	`EObject` which reflects the model element of a development domain apart from the Integration Model
`→baseSource`	Source element of a specified mapping relation defined in the original model context
`→baseTarget`	Target element of a specified mapping relation defined in the original model context
`→parent`	Parent element which includes the definition of the focused mapping relation

Two types of concepts are introduced in the course of the *trace* package, `IMTraceElements`, and `IMTraceConnectors`, with the `IMTrace` being a special variant of an `IMTraceConnector`. An `IMTraceElement` allows a model element of the Integration Model to specify connections to artifacts of other domain-specific models.

For example, the previously introduced `IMTreeNodes` inherit from `IMTraceElement`, while the `IMTrace` defines the concrete connection. In particular, the original connection (`baseSource` and `baseTarget`) is recorded as well, in case it is a connection resulting from a model transformation (see section 7.2.2). The interaction of both constructs makes it possible to annotate structural relationships across domains on `IMComponents` and `IMFunctionalities`. Moreover, properties can be used to map concrete behavior across domains, discussed in section 7.1.3. In particular, this mapping of behavioral descriptions enables the abstract test case execution at the model-level, which is presented throughout chapter 10.

The running example is used to illustrate the concepts just presented. Specifically, figure 7.13 contains examples for mapping structural contents, visualized by the dashed connections between the Integration Model elements and the model elements of other domains. Examples for the mapping of behavioral models or their components are shown in section 7.1.3.

Property Concepts

The `IMTreeNodes` offer the possibility to specify additional information. This can either be data that is not available in the System Model in this form, or data like management metrics, that cannot be mapped to any of the existing modeling domains. Furthermore, more detailed information of the Integration Model is specified in the form of properties (see figure 7.15, table 7.11).

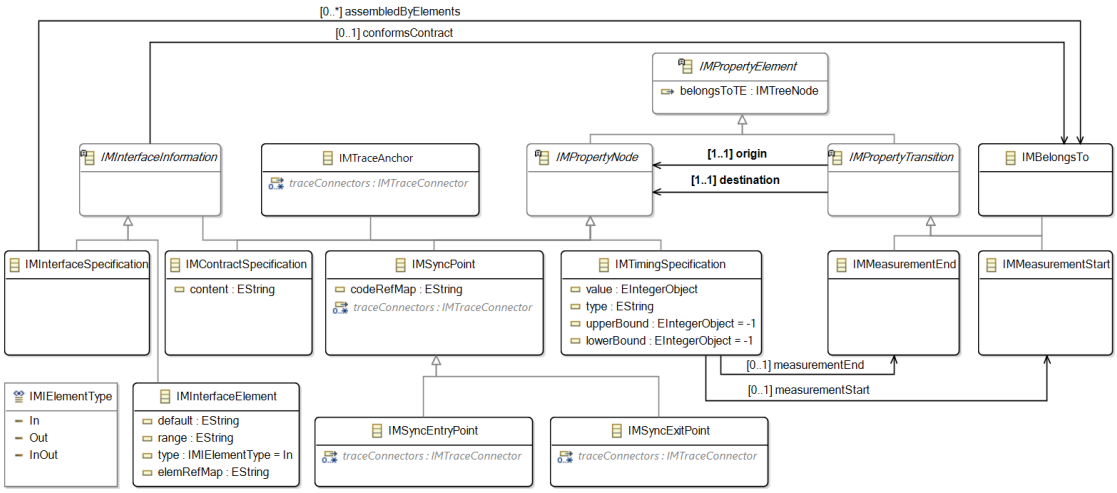


Figure 7.15: Properties package of the IM metamodel

Table 7.11: Metamodel element descriptions for figure 7.15

Concept (Abbreviation)	Description
IMPropertyElement (im_pe)	Abstract concept for implementation purposes of the property package
→belongsToTE	IMTreeNode where the respective property information is attached to
IMPropertyNode (im_pn)	Abstract concept encapsulating specific property information
IMPropertyTransition (im_pt)	Abstract concept for the specification of relations between certain property information
IMBelongsTo (im_bt)	Concept to specify that a property concept belongs to another property concept
IMTraceAnchor (im_ta)	Concept for a more fine-grained specification of mapping information
→traceConnectors	IMTraceConnectors which specify a semantic relation to model elements of other development domains
IMSyncPoint (im_sp)	Concept for detailed mapping of behavioral information across development domains
→codeRefMap	String attribute for the specification of fine-grained mapping based on code fragments
→traceConnectors	IMTraceConnectors which specify a semantic relation to model elements of other development domains
IMSyncEntryPoint (im_sen)	Special version of a IMSyncPoint which reflects the beginning of a mapped behavior sequence
→traceConnectors	IMTraceConnectors which specify a semantic relation to model elements of other development domains
IMSyncExitPoint (im_sex)	Special version of a IMSyncPoint which reflects the end of a mapped behavior sequence
→traceConnectors	IMTraceConnectors which specify a semantic relation to model elements of other development domains
IMTimingSpecification (im_ts)	Concept for the specification of bounds for the execution time between IMSyncPoints
→value	Integer attribute to specify a concrete amount of time
→type	String attribute to determine the unit of the value attribute
→upperBound	Integer attribute to specify the upper bound for the execution time

Table 7.11: Metamodel element descriptions for figure 7.15 (continued)

→lowerBound	Integer attribute to specify the lower bound for the execution time
IMMeasurementStart (im_ms)	Concept to connect the IMTimingSpecification with a certain IMSyncPoint
IMMeasurementEnd (im_me)	Concept to connect the IMTimingSpecification with a certain IMSyncPoint
IMInterfaceInformation (im_ifi)	Concept for attaching contract data to a certain interface
IMInterfaceSpecification (im_ifs)	Concept for the specification of interface information of the SUT
IMInterfaceElement (im_ife)	Concept for the specification of a part of the interface
→default	String attribute for the default value of the interface element
→range	String attribute for the range specification of the interface element
→type	IMIElementType attribute for the current element
→elemRefMap	String attribute for the specification of the referenced System Model element
IMIElementType (im_iet)	Enumeration concept to determine the type of interface information: in, out, inout
IMContractSpecification (im_cs)	Concept to specify contract data
→content	String attribute for the concrete contract information

Like in the other development domains, there are general concepts that inherit attributes, namely IMPropertyElement, IMPropertyNode, and IMPropertyTransition. In particular, a distinction is made between nodes and transitions, with the nodes containing the information and the transitions specifying further-reaching relations between the information. Following the previous *mapping* package, the first concept is the IMTraceAnchor, which allows a more detailed specification of the cross-domain relations. The already known IMTraceConnectors are used. In the context of cross-domain mapping the concept of the IMSyncPoint (IMSyncEntryPoint/IMSyncExitPoint) allows the linking of behavioral models. This concept is intended to synchronize different behavior descriptions, whereby the different types specify the position in the sequence in more detail (entry, intermediate, exit). If estimations of the temporal behavior on the intended target platforms are already available, these can be linked to the IMSyncPoints via the IMTimingSpecification concept (see IMMeasurementStart and IMMeasurementEnd). This allows estimations regarding temporal behavior, which usually cannot be carried out on PIM models. Furthermore, there are concepts for handling interface information enriched with aspects of Contract-Based Design (CBD).

To illustrate these concepts, another part of the running example is presented, namely the internals of the `IMFunctionality CSM_ON Control`. Figure 7.16 contains two `IMSyncPoints` which are about to map behavioral concepts of the System Model with concepts of the Test Model. The mentioned `IMTimingSpecification` property is included, which enables the user together with the measurement connectors, to draw temporal conclusions while testing the respective part of the CSM.

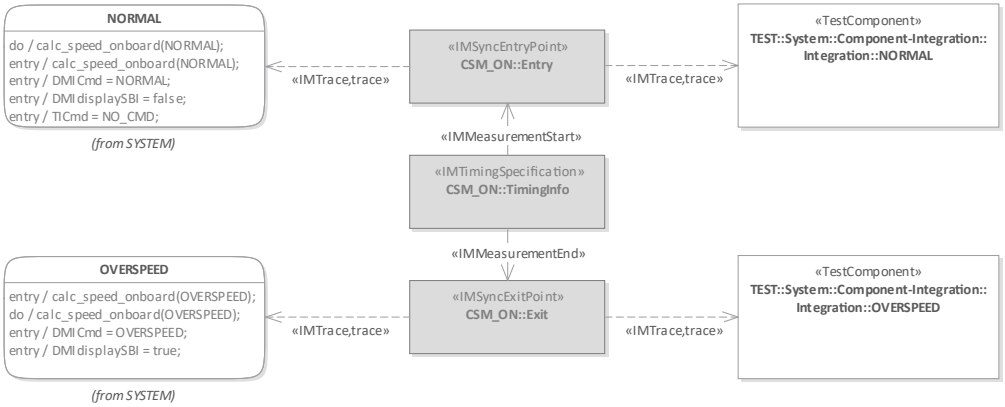


Figure 7.16: Integration Model excerpt for the `CSM_ON Control` sub-system

The Aspect Concept

Besides the extension of concepts of the tree structure by the already presented properties, there is a concept for mapping quantifiable characteristics, annotated to the existing tree structure. This annotation can be used in further processing steps to e.g. specify decisions, depending on empirical values. From a technical point of view, these so-called *Aspects* are further divided into two types. The *intrinsic* aspects, which encapsulate existing data from connected domains for uniform handling. The *synthetic* aspects, which represent already processed model data or data from domains that are not explicitly modeled. Both types of aspects must be defined initially, such that they can be used in further steps. Note, that there is no conceptual relation between our term *Aspect* and concepts like Aspect-Oriented Programming (AOP) or Aspect-Oriented Modeling (AOM).

The Aspect Definition Language represents the interface to the user and fills the concepts provided in the integration metamodel with values. The ANTLR grammar can be seen in listing 7.1 giving an overview of the available constructs. Each aspect is defined by a unique name (`aname`), used to handle the aspect later. The second step is to define the values of the aspect. A distinction is made between aspects with a continuous range of values (`adranged`), aspects with a set of discrete values (`adset`) and aspects based on attributes of concepts from other domains (`adlinkeddef`). For the first two types, only a small selection of data types is explicitly supported but can be

extended. In this way, any number of aspects can be supported in the context of an Integration Model.

```
addsl:
  (aspect ';')*;

aspect:
  aname=ID ':' addefpart;

addefpart:
  (adatatype=DTYPE ':ranged ' adranged | adatatype=DTYPEALL ':set ' adset
  | 'linked ' adlinkeddef );

adranged:
  ( '[' ( min=NUMBER ',' max=NUMBER ) ']' ) | '[' ']';

adset:
  ( '[' ((value=ANYID ',')* value=ANYID) ']' ) | '[' ']';

adlinkeddef:
  eMMName=ID ':' eClassName=ID ':' eAttributeName=ID;

DTYPE: ('Float' | 'Integer');
DTYPEALL: ('String' | 'Float' | 'Integer');
fragment LETTER: [a-zA-Z\u0080-\u00FF_];
fragment DIGIT: [0-9];

ID: (LETTER(LETTER|DIGIT)* | NUMBER);
ANYID: '\'' ( LETTER | DIGIT | '.')+ '\'';

NUMBER: '-'? ('.' DIGIT+ | DIGIT+ ('.' DIGIT*)? );

BLOCK_COMMENT: '/*' .*? '*/' -> channel(HIDDEN);

LINE_COMMENT: '//' ~[\r\n]* -> channel(HIDDEN);

WS: [ \t\r\n]+ -> skip ; // skip spaces, tabs, newlines
```

Listing 7.1: ANTLR grammar for the Aspect Definition Language

The Aspect Specification Language is the counterpart of the previously presented Aspect Definition Language. It specifies explicit values of the aspects at `IMTreeNodes` of the Integration Model. Therefore, a small ANTLR grammar was defined (see listing 7.2), which fills the metamodel concept with values and represents the user interface. In principle, only a set of key-value pairs is implemented, whereby several values can occur on one aspect (`avalues`).

A possible example of an aspect is the risk assessment of a component of the SUT to derive a concrete risk assessment for the respective component from preceding analyses, which can subsequently be used in the context of an aspect for the application of RBT.

```
asdsl:
  (aspect ';')*;

aspect:
  aname=ID ' = ' avalues;

avalues:
  '[' (value=ANYID ',')* value=ANYID ']';

fragment LETTER: [a-zA-Z\u0080-\u00FF_];
fragment DIGIT: [0-9];

ID: (LETTER(LETTER|DIGIT)* | NUMBER);
ANYID: '\'' ( LETTER | DIGIT | '.')+ '\'';

NUMBER: '-'? ('.' DIGIT+ | DIGIT+ ('.' DIGIT*)? );

BLOCK_COMMENT: '/*' .*? '*/' -> channel(HIDDEN);

LINE_COMMENT: '//' ~[\r\n]* -> channel(HIDDEN);

WS: [ \t\r\n]+ -> skip ; // skip spaces, tabs, newlines
```

Listing 7.2: ANTLR grammar for the aspect specification language

In the context of the running example, especially in its Integration Model, both concepts are applied. Therefore, the available aspects and their value sets are specified for the entire Integration Model, namely three aspects, the first of which defines a concrete value set and the last of which defines value ranges for the concrete instances.

```
partID:String:set ['CSM','RSM','TSM'];
safetyLVL:Integer:ranged [0,5];
devPRIO:Integer:ranged [0,10];
```

Based on this definition of aspects, all aspect specifications annotated to the elements of the tree structure are built up. An example is provided, where the specification for the `IMComponent CSM` is shown. For example, as mentioned earlier, the value of `safetyLVL` could be derived from an appropriate safety consideration of the proposed SUT.

```
partID = ['CSM'];
safetyLVL = ['3'];
```

(Modeling) Guidelines & Best Practices

As with the other modeling domains presented in the previous sections, some guidelines apply to the Integration Model, which leads to noticeable improvements in the results of subsequent processing steps. However, it depends on the development approach the Omni Model is used in. On the one hand, development can be driven by

System Modeling. In this case, the structural design of the Integration Model can be completely derived from the available instance information. On the other hand, the Integration Model can be used for the first sketches of the structural breakdown of the system, from which the System Model is subsequently derived. Here, it is particularly important to ensure that the decomposition of the structure provides a sufficient number of integration stages. This is particularly useful and necessary concerning testability, which was mentioned in the modeling guidelines in section 7.1.1. This approach is complementary to the synthetic design concepts of CBD. Thus, each integration level not explicitly provided reduces the transparency of the relationships between the linked domains and thus increases the complexity of the processing steps.

In addition to the structural model elements of the Integration Model, there are several concepts to be considered in the context of behavioral aspects, especially with `IMTraces` and `IMSyncPoints`. As already explained, these model elements are used to establish links across domain boundaries. When linking structural elements, it is important to create and maintain a mapping that is as complete as possible. Linking behavioral descriptions, however, demands careful modeling. An almost complete linkage about the entry points (`IMSyncEntryPoint`) and the possible endpoints (`IMSyncExitPoint`) has to be strived for and maintained. Additional synchronization points are beneficial for some processing steps (see section 10.2.1) but rather disadvantageous for others (see section 10.2.2). However, other concepts of the *properties* package are highly dependent on the level of detail of the modeling, thus no concrete guidelines can be given here.

Finally, some guidelines are given regarding the targeted use of the aspects. Since the aspects represent in particular an essential interface to the user, attention should be paid to the comprehensibility and interpretability of the stored information when defining them. That is, an incomprehensible specification in the value range of the synthetic aspects should be avoided. Further, a significant number of aspects with quite different focus enables a differentiated implementation of further steps.

7.2 Analysis-Specific Models

Starting from a fully specified Omni Model for the respective development context, this section focuses on the subsequent automated processing of model data. Especially against the background of the automated implementation of the STLC steps in the model context, the information around the Test Model and the System Model has to be prepared appropriately.

At this point, horizontal exogenous Model-to-Model Transformation (M2MT) is used, which transforms the different original models of the mentioned domains according to a uniform representation (see figure 7.17). On the one hand, this ensures that the processing chain is independent of the metamodels used in the context of the Integrated Model Basis. On the other hand, the model transformations, which are developed and updated together with the domain experts, ensure that the semantics in the target modeling language meet the expectations. Furthermore, the connections between the partic-

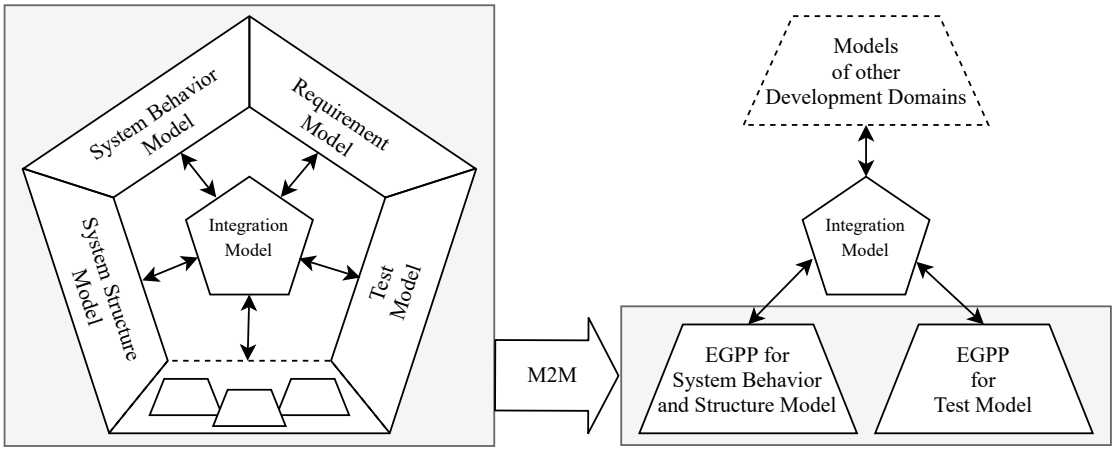


Figure 7.17: Analysis-specific model artifacts in the context of the Omni Model approach

ipating domains specified in the context of the Omni Model are retained or transferred in the context of the M2MT. The connections to non-transformed model artifacts of additional development domains are preserved.

In the following sections the detailed specification of our internal analysis-specific metamodel including execution semantics is rolled out. In a second step, the M2MTs for the system and test domain a further detailed.

7.2.1 Execution Graph++ Metamodel

Apart from the domain-specific models taking part in the Omni Model approach, a domain-independent representation is needed. The use of an internal representation decouples the algorithmic realization of the processing steps of the MCSTLC from concrete instances of the Omni Model. Driven by the overarching objectives defined in section 1.2 the following requirements for the internal computation model were derived:

- ability to reflect structural characteristics of source models, e.g. containment hierarchies
- ability to represent behavioral descriptions, e.g. activity charts or statemachines
  - capture control flow information (potentially incomplete or modeled on different levels of abstraction)
  - capture data flow information (potentially incomplete or specified in different source languages)
- support model evolution as the degree of concreteness steadily rises in development

To meet these requirements and serve as a flexible mechanism for subsequent processing steps of the MCSTLC like e.g. the abstract execution of test cases against the System Model, we developed a concept called Execution Graph++ (EGPP). Following the goal of making this mechanism accessible to a widespread set of application scenarios, we first define the metamodel, which marks the conceptual basis.

Concepts of the Execution Graph++ Metamodel (EGPPMM)

In this section, the components of the underlying metamodel are explained in more detail. Figure 7.18 gives an overview and shows the relationships between the constructs, which are described in more detail below.

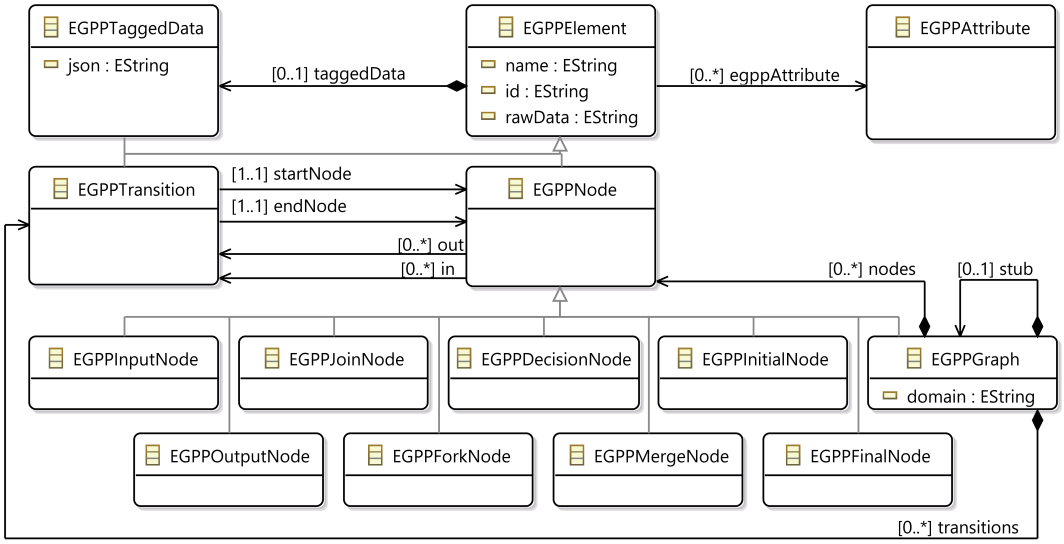


Figure 7.18: Execution Graph++ (EGPP) metamodel

EGPPElement (`el`) is the most abstract concept of the Execution Graph++ Metamodel (EGPPMM), to determine the domain of the concept. It marks the super element of every model artifact of the EGPPMM.

Superclass	
-	
Attribute:Type	**Description**
`name:EString`	name of model element
`id:EString`	unique identifier
`taggedData:EGPPTaggedData`	see `EGPPTaggedData`
`egppAttribute:Collection<EGPPAttribute>`	see `EGPPAttribute`
OCL Invariants	
-	

EGPPGraph (gr) represents the container element for the graph structure. A graph internally maintains the system state, which is given by the set of know variables and the assigned values. It may be part of another `EGPPGraph`, thereby representing a subgraph/-container.

Superclass	
`EGPPElement`	
Attribute:Type	**Description**
`nodes:Collection<EGPPNode>`	set of nodes included (exclusive subgraph's contents)
`transitions:Collection<EGPPTransition>`	set of transitions included (exclusive subgraph's contents)
`stub:EGPPGraph`	reduced version of original (context-dependent)

OCL Invariants

```
context EGPPGraph inv:
self.nodes->select(n | n.oclIsTypeOf(EGPPInitialNode))->size() == 1
context EGPPGraph inv:
self.nodes->select(n | n.oclIsTypeOf(EGPPFinalNode))->size() == 1
```

EGPPNode (no) marks the most general type of node of the graph structure.

Superclass	
`EGPPElement`	
Attribute:Type	**Description**
`in:Collection<EGPPTransition>`	set of incoming transitions
`out:Collection<EGPPTransition>`	set of outgoing transitions

OCL Invariants

-

EGPPTransition (tr) is the most general type of transition included in the graph. Transitions in the EGPP context are always *directed*.

Superclass	
`EGPPElement`	
Attribute:Type	**Description**
`startNode:EGPPNode`	start node of transition
`endNode:EGPPNode`	end node of transition

OCL Invariants

```
context EGPPGraph inv:
self.transitions->forAll(t | t.startNode->notEmpty()
and t.endNode->notEmpty())
```

EGPPInitialNode (`in`) is a special node which represents the start of the control flow captured by the graph structure.

Superclass	
`EGPPNode`	
Attribute:Type	**Description**
-	-

OCL Invariants

```
context EGPPInitialNode inv:
self.in->isEmpty()
context EGPPInitialNode inv:
self.taggedData.codeFragements()->isEmpty()
```

EGPPFinalNode (`fn`) is a special node which represents the end of the control flow captured by the graph structure.

Superclass	
`EGPPNode`	
Attribute:Type	**Description**
-	-

OCL Invariants

```
context EGPPFinalNode inv:
self.out->isEmpty()
context EGPPFinalNode inv:
self.taggedData.codeFragements()->isEmpty()
```

EGPPInputNode (`ipn`) marks a node that is syntactically equivalent to its superclass `EGPPNode`. Semantically, it is a node that consumes/reads information. For example, in the Test Model context, this might be a verification point that checks for a certain attribute value of the System Model, i.e. a system state.

EGPPOutputNode (`opn`) marks a node that is syntactically equivalent to its superclass `EGPPNode`. Semantically, it is a node that emits/writes information. For example in the Test Model context, this might reflect an assignment to a variable of the System Model, i.e. emitting a stimulus.

EGPPDecisionNode (dn) is a node representing a split of the control flow constrained by guards specified along the EGPPTransitions. Further exactly one of the outgoing transitions may be taken at a time, permitting the processing of two paths in parallel.

Superclass	
EGPPNode	
Attribute:Type	**Description**
–	–
OCL Invariants	

```
context EGPPDecisionNode inv:  self.out->size() > 1
```

EGPPMergeNode (mn) is a node representing a merge of the control flow previously split by an EGPPDecisionNode.

Superclass	
EGPPNode	
Attribute:Type	**Description**
–	–
OCL Invariants	

```
context EGPPMergeNode inv:  self.in->size() > 1
```

EGPPForkNode (fon) represents a fork of the control flow into parallel flows.

Superclass	
EGPPNode	
Attribute:Type	**Description**
–	–
OCL Invariants	

```
context EGPPForkNode inv:  self.out->size() > 1
```

EGPPJoinNode (jon) is a node representing a join of the control flow, while the parallel flows need to be spawned at the same EGPPForkNode.

Superclass	
EGPPNode	
Attribute:Type	**Description**
–	–
OCL Invariants	

```
context EGPPJoinNode inv:  self.in->size() > 1
```

Further, the number and instances of spawning paths in an `EGPPForkNode`, later on, joined in an `EGPPJoinNode` need to be equal. To define such a constraint, we utilize a DataFlow Analysis (DFA)-based specification with attribute grammars as per Saad et al. [152].

Therefore, we define the attribute `numParflows` representing a list of integers that indicate the number of parallel EGPP-paths currently active at a time. Lists with more than one entry indicate the existence of parallel regions within a parallel region. Thereby, the last entry represents the most inner region. A list entry equal to zero indicates a malformed `EGPPGraph`, due to non-conformance to the criteria specified at the beginning of this paragraph. Listing 7.3 shows the production rules for the initialization of the `numParflows` attribute of the different types of nodes previously introduced.

```
attribution parflows_analysis {
  -- attribute that indicates the number of parallel flows currently active
  attribute assignment numParflows : OCLSequence
    initWith { };

  -- rule to check how many parflows are active at the direct predecessor
  rule ocl node_parflows : standard
    "return self.in.startNode.numParflows()";

  extend egpp_no with {
    occurenceOf numParflows calculateWith node_parflows;
  }

  -- rule to check how many parflows are active at the direct predecessor
  rule ocl forknode_parflows : standard
    "return self.in.startNode.numParflows().append(self.out->size())"

  extend egpp_fon with {
    occurenceOf numParflows calculateWith forknode_parflows;
  }

  -- rule to check how many parflows are active at the direct predecessor
  rule ocl joinnode_parflows : standard
    "def: ref:Integer = self.in->first().numParflows()
    if (self.in->any(t : t.startNode.numParflows()->last() = 0)) then
      return self.in.startNode.numParflows().append(0)
    else
      if (self.in->forAll(t : t.startNode.numParflows()->last() = ref)) then
        return self.in.startNode.numParflows()
      ->subSequence(1, self.in.startNode.numParflows()->size()-1))
      endif
    endif"

  extend egpp_jon with {
    occurenceOf numParflows calculateWith joinnode_parflows
  }

  extend egpp_in with {
    occurenceOf numParflows calculateWith {1};
  }
}
```

Listing 7.3: The attribution for the `parflows_analysis`

In a second step, based on the initialized attributes, the validity of the EGPPGraph can be checked with the following analysis (see listing 7.4). The last entry of the numParflows list is checked for the EGPPFinalNode.

```
attribution parflows_validation {
  -- constraint that indicates if the EGPPGraph
  -- is well-formed concerning the parallel regions
  attribute constraint checkParflows : error
    "EGPPGraph contains malformed parallel regions";

  -- use the result of numParflows
  rule ocl egpp_fn_checkParflows : standard
    "return self.numParflows()->last() = 1";

  -- attach the 'checkParFlows' to the EGPPFinalNode(egpp_fn)
  -- computed with the rule 'egpp_fn_checkParflows'
  extend egpp_fn with {
    occurenceOf checkParflows calculateWith egpp_fn_checkParflows;
  }
}
```

Listing 7.4: The attribution for the parflows_validation

The list of already introduced concepts together with the two data flow analyses just presented defines valid structures in the context of EGPP implementation. However, what is not shown here is the information that extends the previous purely structural view to the data flow level. For this purpose, further concepts are introduced, which are explained in the following.

EGPPTaggedData (td) represents any kind of uninterpreted data specified alongside an EGPPElement.

Superclass	
EGPPElement	
Attribute:Type	**Description**
json:EString	uninterpreted JSON-String
OCL Invariants	
-	

During the exogenous horizontal M2MT from a source model to an EGPP model, the until now unmentioned data flow information is captured by an EGPPTaggedData object. To improve flexibility and extendability, the data is serialized as a JavaScript Object Notation (JSON) string with a node named Code. Thereby, the Code node supports the concepts declaration $\langle DECL \rangle$, statement $\langle STMT \rangle$, condition $\langle COND \rangle$, and function (FUNC) in a pseudocode-like syntax (see Grammar 7.1). Further, the usage of $\langle DECL \rangle$, $\langle STMT \rangle$ or $\langle FUNC \rangle$ code fragments is only allowed in EGPPTaggedData (td) annotated to EGPPNode (no) or any inheriting concepts. In contrast, the $\langle COND \rangle$ fragments may be used inside EGPPTaggedData (td) of any EGPP metamodel element.

⟨*CODE*⟩ ::= ⟨*DECL*⟩ | ⟨*STMT*⟩ | ⟨*COND*⟩ | ⟨*FUNC*⟩

⟨*DECL*⟩ ::= 'int' ⟨*IDENT*⟩ '=' ⟨*INTEGER*⟩ ';'
| 'double' ⟨*IDENT*⟩ '=' ⟨*DOUBLE*⟩ ';'
| 'string' ⟨*IDENT*⟩ '=' ⟨*STRING*⟩ ';'

⟨*STMT*⟩ ::= ⟨*IDENT*⟩ '=' ⟨*EXP*⟩ ';'
| ⟨*IDENT*⟩ '=' ⟨*FUNC*⟩ ';'
| ⟨*STMT*⟩ ⟨*STMT*⟩

⟨*EXP*⟩ ::= ⟨*IDENT*⟩
| ⟨*CONST*⟩
| ⟨*EXP*⟩ '+' ⟨*EXP*⟩
| ⟨*EXP*⟩ '-' ⟨*EXP*⟩
| ⟨*EXP*⟩ '·' ⟨*EXP*⟩
| ⟨*EXP*⟩ '/' ⟨*EXP*⟩

⟨*COND*⟩ ::= 'true' | 'false'
| ⟨*IDENT*⟩
| ⟨*CONST*⟩
| '!' ⟨*COND*⟩
| ⟨*COND*⟩ '&&' ⟨*COND*⟩
| ⟨*COND*⟩ '||' ⟨*COND*⟩
| ⟨*COND*⟩ '==' ⟨*COND*⟩
| ⟨*COND*⟩ '!=' ⟨*COND*⟩
| ⟨*COND*⟩ '<' ⟨*COND*⟩
| ⟨*COND*⟩ '>' ⟨*COND*⟩
| ⟨*COND*⟩ '≤' ⟨*COND*⟩
| ⟨*COND*⟩ '≥' ⟨*COND*⟩

⟨*FUNC*⟩ ::= ⟨*IDENT*⟩ '(' ⟨*ARGLIST*⟩ ')' ';'

⟨*ARGLIST*⟩ ::= (⟨*IDENT*⟩ | ⟨*CONST*⟩) * (',' ⟨*IDENT*⟩ | ⟨*CONST*⟩) *

⟨*CONST*⟩ ::= ⟨*INTEGER*⟩ | ⟨*DOUBLE*⟩ | ⟨*STRING*⟩

Grammar 7.1: Grammar for the `Code` fragments of `EGPPTaggedData`

By the grammar shown, variables can be mapped with three different data types, namely `int`, `double`, and `string`. The possible expressions can further be constructed by the basic mathematical operators `+`, `-`, ·, and `/`. There are some restrictions on the conditions. Basic boolean and relational operators can be used to create conditions based on variables, constants, and the two possible truth values `true` and `false`. These constructs can be used to map a large part of the elementary code constructs used, which is, among other things, the basis for further analyses on these model artifacts.

In addition to the data flow information, it is intended to store uninterpreted data in a

JSON node called *Comment*. Under certain circumstances, this data can be used for further processing or interpretation in subsequent steps. A possible application scenario for this can be seen in section 7.2.2. Further, a concept related to the transformation from uninterpreted to interpreted data is given by the `EGPPAttribute`.

EGPPAttribute (`at`) is meant to capture interpreted internal information which is attached to an `EGPPElement`.

With this model element, calculations can be outsourced to a pre-processing for certain analyses, which may involve computationally intensive steps. The pre-processing fills appropriate `EGPPAttributes` for later execution of the analysis.

Semantics of the EGPPMM

Based on the EGPP metamodel concepts and the previously defined language fragments, we define the semantics of the EGPPMM, which represents the basis for the subsequent specification of a model interpreter in the context of ATE. First, we focus on the semantics of code fragments syntactically conforming to the previous grammar.

Definition 29 (**Datatypes and Value Ranges**)
Variables or constants in the context of language artifacts specified before, support the datatypes $\langle INTEGER \rangle$, $\langle DOUBLE \rangle$, *and* $\langle STRING \rangle$. *The respective type identifiers build up the set* $\Pi = \{int, double, string\}$. *Further, we define the value ranges linked to the type identifiers as follows:*

$$\mathbb{V}_t = \begin{cases} \mathbb{Z} & \text{if } t = int \\ \mathbb{R} & \text{if } t = double \\ \mathbb{S} & \text{if } t = string \end{cases}$$

$$\text{with } t \in \Pi;$$

$$\mathbb{Z} = \text{ the set of integer values between INT\_MIN and INT\_MAX;}$$

$$\mathbb{R} = \text{ the set of floating numbers between DOUBLE\_MIN and DOUBLE\_MAX;}$$

$$\mathbb{S} = \text{ the UTF-8 representation of a string value;}$$

Computations with `'string'` *type are based on the UTF-8 values,*
while comparisons are based on the length of the string

Definition 30 (**Variable Types**)
A variable holds a certain type of information Π *as previously introduced. This type may further be revealed as follows:*

$$\tau : Var \rightarrow \Pi$$

Definition 31 (**Variable Valuations**)
Variables determine certain states $\Sigma : Var \rightarrow \mathbb{V}$*, while the variable's current value is specified by the following mapping function:*

$$\sigma_t : Var \rightarrow \mathbb{V} \text{ with } t \in \Pi$$

Let $\sigma(x)$ *be a function to access the variable's current value and*

$$\sigma[x_t \rightarrow v_t] \text{ to update the value of } x_t \text{ with } v_t$$

$$\text{with } x_t \in Var_t \; ; \; v_t \in \mathbb{V}_t$$

Based on the previous definitions, the update of a variable leads to a change of its valuation, which conforms to the following equation.

$$\sigma[x_t \rightarrow v_t](x'_t) = \begin{cases} \sigma(x'_t) & \text{if } x_t \neq x'_t \\ v_t & \text{if } x_t = x'_t \end{cases} \tag{7.1}$$

Following the basic definitions, the compositional semantics of the $\langle DECL \rangle$ and $\langle STMT \rangle$ fragments of our grammar (see 7.1) are specified. To specify the semantics of declarations and valuation expressions of our grammar, we define the semantic function $\mathcal{I}$.

Declaration 1 (**Semantic Function** $\mathcal{I}$)
This function defines the transition between variable states Σ *as follows:*

$$\mathcal{I}[\![-]\!] : \langle DECL \rangle \cup \langle STMT \rangle \rightarrow (\Sigma \rightarrow \Sigma)$$

To specify the semantics of declarations and valuation expressions of our grammar, we further define the semantic function $\mathcal{A}$.

Declaration 2 (**Semantic Function** $\mathcal{A}$)
This function defines the projection of a state to a certain value range as follows:

$$\mathcal{A}[\![-]\!] : \langle EXP \rangle \rightarrow (\Sigma \rightarrow \mathbb{V}_t)$$

$$\text{with } t \in \Pi$$

Before we start with further semantic specifications, we define w.l.o.g. the following elements to be representatives of the respective category of our grammar tags with a new index determining a new representative:

$$e \in \langle EXP \rangle \; ; \; c \in \langle COND \rangle \; ; \; i \in \langle IDENT \rangle \; ; \; s \in \langle STMT \rangle \; ; \; d \in \langle DECL \rangle \; ; \; k \in \langle CONST \rangle$$

and p being a prefix of the string representation of a declaration

Furthermore, in the context of the following definitions, any mathematical and relational operators are used in their original semantics.

$$\mathcal{I}[\![pi = k]\!]\sigma = \sigma[i \to k] \text{ with } \begin{cases} \tau(k) = int & \text{if } p = \text{'int'} \\ \tau(k) = double & \text{if } p = \text{'double'} \\ \tau(k) = string & \text{if } p = \text{'string'} \end{cases} \tag{7.2}$$

$$\mathcal{I}[\![i = e]\!]\sigma = \sigma[i \to z] \text{ with } z = \mathcal{A}[\![e]\!]\sigma \tag{7.3}$$

$$\mathcal{I}[\![s_1; s_2]\!]\sigma = \mathcal{I}[\![s_2]\!]\sigma' \text{ with } \sigma' = \mathcal{I}[\![s_1]\!]\sigma \tag{7.4}$$

$$\mathcal{A}[\![i]\!]\sigma = \sigma(i) \tag{7.5}$$

$$\mathcal{A}[\![k]\!]\sigma = v_t \text{ with } v_t \text{ being the value of the constant } k \text{ of type } t \in \Pi \tag{7.6}$$

$$\begin{gathered} \mathcal{A}[\![e_1 + e_2]\!]\sigma = \mathcal{A}[\![e_1]\!]\sigma + \mathcal{A}[\![e_2]\!]\sigma \\ \text{if } \tau(\mathcal{A}[\![e_1]\!]\sigma) = \tau(\mathcal{A}[\![e_2]\!]\sigma) \wedge \tau(\mathcal{A}[\![e_1]\!]\sigma) \in \{int, double\} \end{gathered} \tag{7.7}$$

The analogous continuation of the equation (7.7) for the remaining binary mathematical operators $\{-, \cdot, /\}$ is analogous. Further, there are no semantic rules regarding structural aspects like sequences, branches, or loops because they are not covered by the language fragments, but by the EGPP nodes and transitions.

In addition to the semantic functions $\mathcal{I}$ and $\mathcal{A}$ for declarations and valuation expressions, we further define a semantic function $\mathcal{B}$ for the boolean expressions.

Definition 32 (**Semantic Function** $\mathcal{B}$)
We define the semantic function $\mathcal{B}$ as follows:

$$\mathcal{B}[\![-]\!] : \langle COND \rangle \to (\Sigma \to \mathbb{B})$$

$$\textit{with } t \in \Pi \textit{ and } \mathbb{B} = \{tt, ff\};$$

$$\mathcal{B}[\![true]\!]\sigma = tt \tag{7.8}$$

$$\mathcal{B}[\![false]\!]\sigma = ff \tag{7.9}$$

$$\mathcal{B}[\![i]\!]\sigma = \begin{cases} tt & \text{if } (\tau(i) = int \wedge \sigma(i) \neq 0) \vee \\ & (\tau(i) = double \wedge \sigma(i) \neq 0.0) \vee \\ & (\tau(i) = string \wedge \sigma(i) \neq \text{'0'}) \\ ff & \text{otherwise} \end{cases} \tag{7.10}$$

$$\mathcal{B}[\![k]\!]\sigma = \begin{cases} tt & \text{if } (\tau(k) = int \wedge k \neq 0) \vee \\ & (\tau(k) = double \wedge k \neq 0.0) \vee \\ & (\tau(k) = string \wedge k \neq \text{'0'}) \\ ff & \text{otherwise} \end{cases} \tag{7.11}$$

$$\mathcal{B}[\![!c]\!]\sigma = \neg\mathcal{B}[\![c]\!]\sigma \tag{7.12}$$

$$\mathcal{B}[\![c_1 \&\& c_2]\!]\sigma = \mathcal{B}[\![c_1]\!]\sigma \wedge \mathcal{B}[\![c_2]\!]\sigma \tag{7.13}$$

$$\mathcal{B}[\![c_1 || c_2]\!]\sigma = \mathcal{B}[\![c_1]\!]\sigma \vee \mathcal{B}[\![c_2]\!]\sigma \tag{7.14}$$

$$\mathcal{B}[\![c_1 == c_2]\!]\sigma = \begin{cases} tt & \text{if } \mathcal{B}[\![c_1]\!]\sigma = \mathcal{B}[\![c_2]\!]\sigma \\ ff & \text{if } \mathcal{B}[\![c_1]\!]\sigma \neq \mathcal{B}[\![c_2]\!]\sigma \end{cases} \tag{7.15}$$

$$\mathcal{B}[\![c_1 != c_2]\!]\sigma = \neg\mathcal{B}[\![c_1 == c_2]\!]\sigma \tag{7.16}$$

$$\mathcal{B}[\![c_1 < c_2]\!]\sigma = \begin{cases} tt & \text{if } \mathcal{A}[\![c_1]\!]\sigma < \mathcal{A}[\![c_2]\!]\sigma \wedge \\ & \tau(c_1) = \tau(c_2) \\ ff & \text{otherwise} \end{cases} \tag{7.17}$$

$$\mathcal{B}[\![c_1 > c_2]\!]\sigma = \begin{cases} tt & \text{if } \mathcal{A}[\![c_1]\!]\sigma > \mathcal{A}[\![c_2]\!]\sigma \wedge \\ & \tau(c_1) = \tau(c_2) \\ ff & \text{otherwise} \end{cases} \tag{7.18}$$

$$\mathcal{B}[\![c_1 <= c_2]\!]\sigma = \neg\mathcal{B}[\![c_1 > c_2]\!]\sigma \tag{7.19}$$

$$\mathcal{B}[\![c_1 >= c_2]\!]\sigma = \neg\mathcal{B}[\![c_1 < c_2]\!]\sigma \tag{7.20}$$

Apart from the semantics of the code fragments included in our EGPP, there are model elements to specify the control flow structure of the underlying graph. To address some characteristics of the EGPP during the semantic specification, we define some terms in this context.

Definition 33 (**EGPPGraph Characteristics**)

Φ_g *represents the set of graph nodes (*`gr.nodes`*)*

$\Theta_g : \Phi_g \times \Phi_g$ *represents the set of graph transitions (*`gr.transitions`*)*

with $n \in \Phi_g$ *and* $t \in \Theta_g$

$code(n) \in \langle DECL \rangle \cup \langle STMT \rangle \cup \langle COND \rangle$ *to access code fragments of a node n*

$cond(t) \in \langle COND \rangle$ *to access code fragments of a transition t*

Structural Definition of the EGPP Semantics

Apart from the state of variables included in code fragments, a model element as part of the control flow is necessary to determine the overall state of the system. Every EGPPGraph (gr) maintains its systems state, which is given by the currently active control flow-node and the states of the variables known to this point.

Definition 34 (System State of a EGPPGraph)
The system state of a EGPPGraph instance may be defined as follows:

$$\langle n, \sigma \rangle \in \Phi_g \times \Sigma$$

To visually support the definition of a system state, the following excerpt of a `EGPPGraph` structure is given.

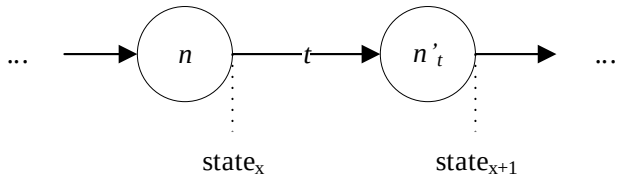


Figure 7.19: An illustration of the different system states ($state_x$ and $state_{x+1}$) in a EGPP flow

The evolution of the system state throughout the execution of the `EGPPGraph` structure conforms to the following step rule:

$$\langle n, \sigma \rangle \xrightarrow{t} \langle n', \sigma' \rangle \quad \text{if} \quad (n, n') \in \Theta_g \wedge \mathcal{I}[\![code(n')]\!]\sigma = \sigma'$$
$$\text{with } t \text{ being the chosen transition between } n \text{ and } n'$$

Based on the definitions in the context of a `EGPPGraph`, we further define the structural-operational semantics of the control flow structure.

$$\langle no, \sigma \rangle \xrightarrow{t} \langle no', \mathcal{I}[\![code(no')]\!]\sigma \rangle \quad \text{if} \quad \mathcal{B}[\![cond(t)]\!]\sigma = tt \qquad (\texttt{no}^{tt})$$
$$\langle no, \sigma \rangle \xrightarrow{t} \langle no, \sigma \rangle \quad \text{if} \quad \mathcal{B}[\![cond(t)]\!]\sigma = ff \qquad (\texttt{no}^{ff})$$

Especially, $\texttt{no}^{ff}$ results in the system stalling in a certain state until a transition t can be satisfied. The rules ($\texttt{no}^{tt}$) and ($\texttt{no}^{ff}$) hold for all of the subtypes of `no` (`EGPPNode`), e.g. `fon` (`EGPPForkNode`), if no special semantic rule is defined. The first exception to this rule is the case for the `dn` (`EGPPDecisionNode`), where a decision has to be taken as to how to proceed. Further, this decision has to be clearly defined by the conditions of the transitions and may in every case only produce one single resulting path.

$$\langle dn, \sigma \rangle \xrightarrow{t_i} \langle dn', \mathcal{I}[\![code(dn')]\!]\sigma \rangle$$
$$\text{if} \quad \mathcal{B}[\![cond(t_i)]\!]\sigma = tt \wedge \forall t \in \{t_1, \ldots, t_n\} \setminus \{t_i\} : \mathcal{B}[\![cond(t)]\!]\sigma = ff \qquad (\texttt{dn})$$

Another exception is the parallel regions in the `EGPPGraph`. At this point, a simplified merging of the parallel developing system states is applied, as per the following definition.

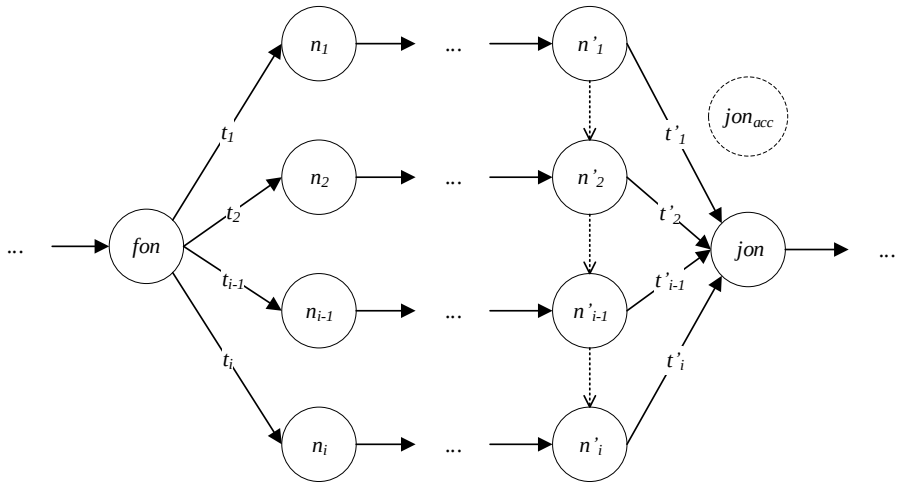


Figure 7.20: A schematic for parallel flows between `EGPPForkNode` (`fon`) and `EGPPJoinNode` (`jon`)

Definition 35 (System State Merge at EGPPJoinNode)
Let $i \in \mathbb{N}$ be the number of parallel flows in a EGPPGraph as per figure 7.20. Further, the end nodes of the parallel flows are ordered about the lexicographic order of the last transition's labels from the last node to `jon`, e.g. t'_1 for the uppermost path of figure 7.20. The joint system state at the jon node can be computed according to the following rules. The dotted lines in figure 7.20 show the iterative computation and the propagation of the accumulated intermediate state across parallel flows.

$$\langle n'_k, \sigma \rangle \xrightarrow{t'_k} \langle jon_{acc}, \sigma_{acc} \rangle \quad \text{if} \quad \mathcal{B}[\![code(t'_k)]\!]\sigma = tt \tag{$\texttt{jon}^{tt}_k$}$$

$$\langle n'_k, \sigma \rangle \xrightarrow{t'_k} \langle jon_{acc}, \sigma \rangle \quad \text{if} \quad \mathcal{B}[\![code(t'_k)]\!]\sigma = ff \tag{$\texttt{jon}^{ff}_k$}$$

$$\langle n'_i, \sigma \rangle \xrightarrow{t'_i} \langle jon, \mathcal{I}[\![code(jon)]\!]\sigma_{acc} \rangle \quad \text{if} \quad \mathcal{B}[\![code(t'_i)]\!]\sigma = tt \tag{$\texttt{jon}^{tt}_i$}$$

$$\langle n'_i, \sigma \rangle \xrightarrow{t'_i} \langle jon, \mathcal{I}[\![code(jon)]\!]\sigma \rangle \quad \text{if} \quad \mathcal{B}[\![code(t'_i)]\!]\sigma = ff \tag{$\texttt{jon}^{ff}_i$}$$

with $jon_{acc} \notin \Phi_g$ representing an accumulator node

and σ either representing the initial valuations, or σ_{acc} from the previous merge step

and σ_{acc} representing the merged valuations of accumulator node

and $k \in \{1, \ldots, i-1\}$

As mentioned in the initial part of this section, `EGPPGraphs` are designed to be part

of another `EGPPGraph`. Essentially, there are two types of scenarios to be considered, entering a sub-graph (gr_{in}) or leaving a sub-graph (gr_{out}). Consequently, the traversal of an `EGPPGraph` conforms to the following set of semantic rules further illustrated in figure 7.21:

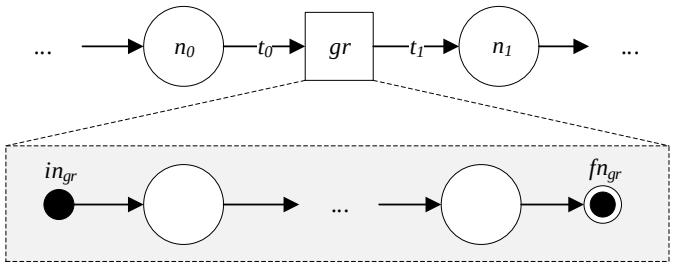


Figure 7.21: An illustration of a sub-graph embedded in a flow of its parent `EGPPGraph` (`gr`)

$$\frac{\langle n_0, \sigma\rangle \xrightarrow{t_0} \langle in_{gr}, \sigma\rangle \quad \text{if } \mathcal{B}[\![cond(t_0)]\!]\sigma = tt}{({gr_{in}}^{tt})}$$

$$\frac{\langle n_0, \sigma\rangle \xrightarrow{t_0} \langle n_0, \sigma\rangle \quad \text{if } \mathcal{B}[\![cond(t_0)]\!]\sigma = ff}{({gr_{in}}^{ff})}$$

$$\frac{\langle fn_{gr}, \sigma\rangle \xrightarrow{t_1} \langle n_1, \sigma\rangle \quad \text{if } \mathcal{B}[\![cond(t_1)]\!]\sigma = tt \wedge fn_{gr} \in \Phi_{gr}}{({gr_{out}}^{tt})}$$

$$\frac{\langle fn_{gr}, \sigma\rangle \xrightarrow{t_1} \langle gr, \sigma\rangle \quad \text{if } \mathcal{B}[\![cond(t_1)]\!]\sigma = ff \vee fn_{gr} \notin \Phi_{gr}}{({gr_{out}}^{ff})}$$

The syntactic and semantic definitions presented here provide the basis for most of the following steps. In particular, this laid the foundation for the Abstract Test Execution explained in more detail in section 10.2.

7.2.2 Model to Model Transformations

Besides the definition of the internal metamodel, which serves to implement the different analyses in the context of the Integrated Model Basis, the M2MTs (see figure 7.17) play an important role. These horizontal exogenous transformations allow different source models to be mapped to the previously introduced metamodel and, in the process, to improve weaknesses of the source metamodel by expert knowledge. However, when specifying the model transformations, care must be taken to ensure that the resulting representation is as accurate as possible to keep the differences between the model and the code representation as small as possible. If the specification is incorrect, the results of the analyses based on it may not provide usable results.

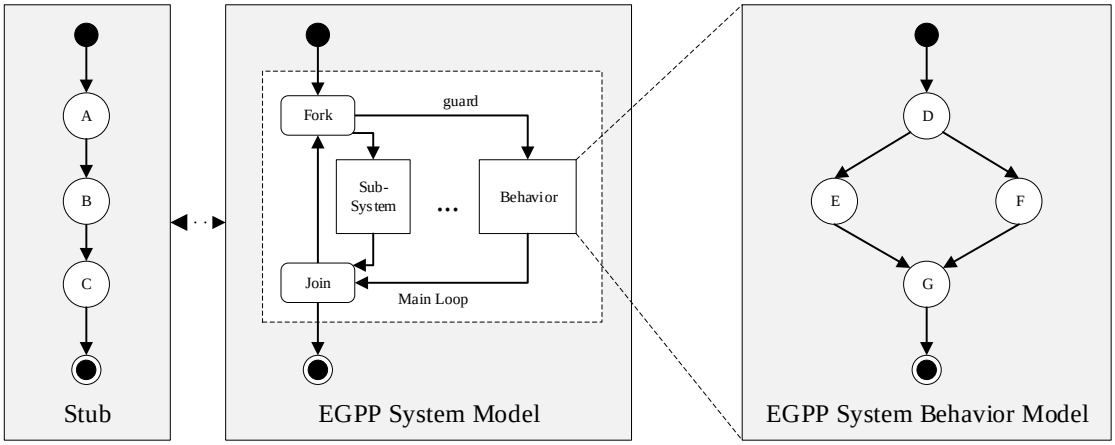


Figure 7.22: Pattern for EGPP instance transformed from structure and behavior System Model artifacts

The transformation is divided into two phases, whereby the structure and control flow aspects as well as the data flow aspects are performed separately. In addition to the differentiation into the phases of the transformation, a distinction is made according to the model context, that is, whether the original model is used for System Modeling or for Test Modeling.

Transformation of Structure and Control Flow Information

First, the transformation of structural information is discussed. For this purpose, patterns are developed for the two modeling contexts *System* and *Test*, according to which the instances of the target modeling language EGPP are built. The following section discusses the set of rules for the transformation of system structure and behavior information.

System Structure and Behavior Information to Execution Graph++ As already described in the introduction of the metamodel concepts of EGPP, both the structure and the behavior of a system can be implemented integrated into the graph-based representation. An exemplary section of an EGPP structure is shown in figure 7.22, where the system level serves as the entry point, which is developed top-down to the smallest included components. In the middle of the figure, there is the `EGPPGraph`, containing especially the *Main Loop* as its main component. This represents the functionality of conventional systems to activate certain components either time- or event-triggered (*guard*), which in this case corresponds to traversing one of the parallel paths including possible sub-graphs. The concepts contained in *Main Loop* can be either subsystems or behavioral models, each of which is mapped using `EGPPGraph`. On the right side of figure 7.22 is an example graph containing two alternative processes, where the system state can be changed by expressions contained in the nodes `D`, `E`, `F`, `G`. On the other

hand, if you take a look at the `EGPPGraph` of the sub-system shown, it would have the same components as the graph shown and explained in the middle of figure 7.22. How often this encapsulation and, accordingly, the nested *Main Loops* occur depends on the number of integration levels of the System Model.

An essential difference between the use of the EGPP metamodel for system and Test Modeling is the use of the so-called *stub*. This is an `EGPPGraph` that is coupled to another `EGPPGraph` and reflects the initialization of its variables relevant to the system state. As shown in figure 7.22 on the left, this is an `EGPPGraph` that consists of a path, with nodes `A`, `B`, `C` containing variable assignments of initial values, e.g. `x = 5;`, while the initial values are not mandatory and may therefore be reduced to declarations. Processing of the information contained in the *stub* is done once at the beginning of execution, not per iteration of the *Main Loops*. The described *stub* concept applies to the sub-system and behavior `EGPPGraphs` of the lower levels.

As in the previous chapters, the running example shows how a portion of the System Model is mapped to the EGPP representation. Figure 7.23 shows the nesting of different EGPP graphs, each of which retains sub-functionality.

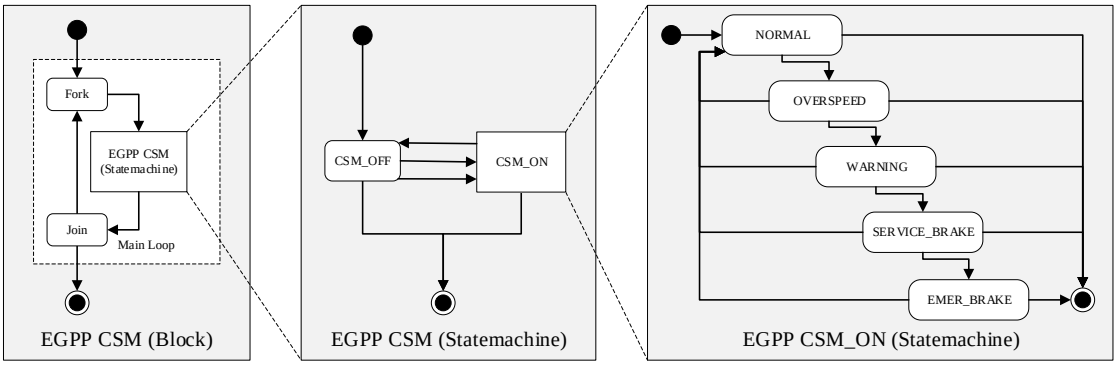


Figure 7.23: Excerpt of the EGPP model for the CSM System Model

For clarity, detailed information such as edge labels has been omitted. On the left side, one can see the original SYSML block CSM which has been transformed into an `EGPP-Graph`. Since the CSM block does not contain any further blocks, only the transformed statemachine (original: see figure 16.3 (Supplementary Material)) is included in the main cycle. In the middle of the picture, you can see the first statemachine in the EGPP context, in which the sub-statemachine (original: see figure 16.4 (Supplementary Material)) shown on the right side is integrated. In the EGPP variants of the two statemachines it is noticeable that an explicit final state was specified in each case, which can be reached from all intermediate states. This is necessary to comply with the EGPP metamodel.

Test Information to Execution Graph++ Analogous to the development of a pattern for software modeling, the structure of the EGPP models, which represents the test

context, is explained. At this point, a schematic representation of the structure of the target model is used, as shown in figure 7.24.

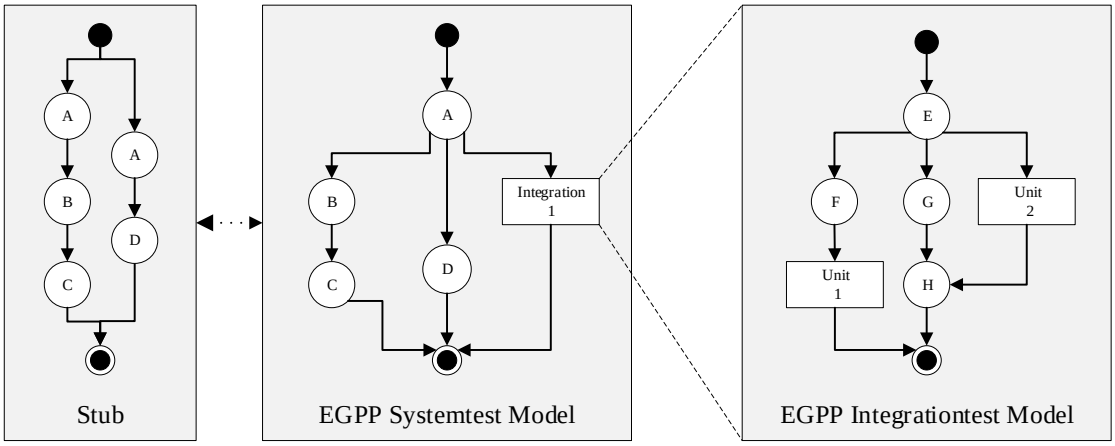


Figure 7.24: Pattern for EGPP instance transformed from Test Model artifacts

As already explained in the foundations' chapter, test cases are specified for different integration levels. Thus, test cases of higher integration levels build on tested components of lower levels. This situation is reflected in EGPP models, where Test Modeling is divided along with the different levels of integration, starting from system level down to unit level. In the middle of figure 7.24, a transformed Test Model for the system level is shown schematically. Three possible paths (test cases) can be derived from this model, where one path incorporates the externally perceivable effects of an intermediate level component *integration 1* on the next lower level. The Test Modeling of this component is shown on the right side of the figure. Based on this model a set of paths (test cases) can be derived, which partly build up again on insights from tests on deeper integration levels (see *Unit 1* and *Unit 2*).

On the left side of figure 7.24, one can see another `EGPPGraph`, showing the role of the so-called *stubs*. In the context of Test Modeling, the *stub* encapsulates a set of paths, while each of the paths contains a set of nodes, that first evaluate the overall system state and then manipulate the system state accordingly. The contents of the paths emerge from successful test executions against the corresponding EGPP model. With sufficient test data, the paths contained in the *stub* can be aggregated to paths that represent equivalence classes to the input data. The information contained in the *stub* model is further utilized in the context of our approach to Abstract Test Execution (see section 10.2.4). Here, the information collected at lower levels of integration provides value to the model interpreter, which can use it to adjust the current system state without further evaluation of embedded EGPP models. Overall, extensive testing starting at lower integration levels can improve the performance of MCSTLC.

In this section, we use the running example, which is intended to illustrate the explained facts. Figure 7.25 illustrates the split of the Test Model into the mentioned integration levels.

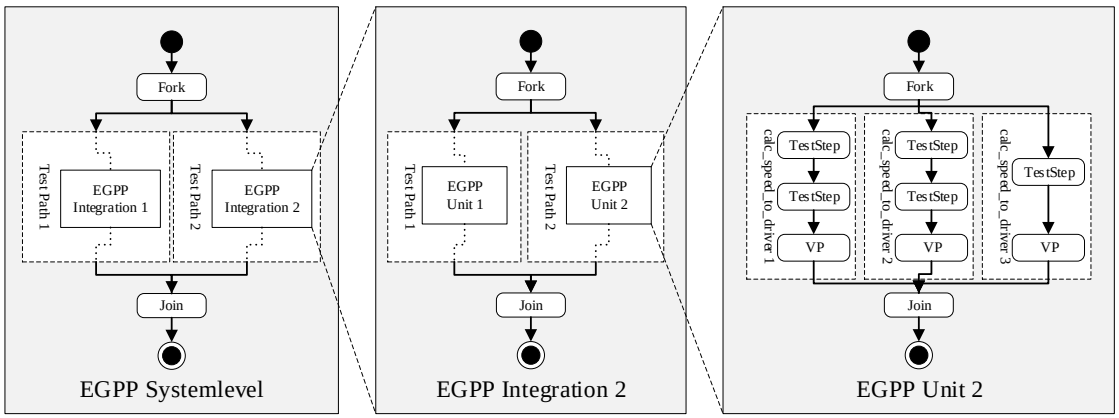


Figure 7.25: Excerpt of the EGPP model for the CSM Test Model

The EGPP representation of the Test Model for the system test level is shown on the left side of the figure. In particular, it represents a set of test cases by providing multiple paths from the `EGPPInitialNode` to the `EGPPFinalNode`. As part of these paths, the stubs of the Test Models of deeper integration levels (see *EGPP Integration 2*) are included. The concept of stubs is dealt with separately in the context of chapter 8 since several independent models are potentially created from one model. This indicates that already tested behavior is used in an extended test context. This can be continued analogously through the intended integration levels up to the unit level. At the right side of the figure, the test cases for the method `calc_speed_to_driver` are shown as an example, which in turn represent individual paths through the Test Model `EGPP Unit 2`. The test cases each consist of `EGPPNodes`, which on the one hand represent *TestSteps* and on the other hand *VerificationPoints (VP)*, determined by the types and contents of the `EGPPTaggedData`.

Transformation of Data Flow Information

In addition to the transformation of the structural information as well as the modeled control flow, the original model can already contain detailed information which has to be transformed. Detailed information such as code fragments, which are embedded in the model artifacts, conform to a certain language. Depending on the language scope and the size of the embedded code fragments, the transformation of the data flow information can have effects on the previously considered structure and control flow transformations.

The aim of the second phase of the M2MT is, to map the control flow components used in code fragments in (sub-)graphs and, to store the atomic data flow information in the `EGPPTaggedData` of the respective model elements. For the former, the transformation procedure is relatively obvious. If the code fragments to be transformed contain control flow information, the model artifact previously implemented as `EGPPNode` is replaced by an `EGPPGraph`, which in turn reflects the code-based control flow. The atomic data

flow information (assignments, conditions, etc.) is then stored as usual in the `EGPPTaggedData` of the respective `EGPPNode` or `EGPPTransition`.

For the latter, depending on the programming language used, the concepts need to be mapped to our pseudocode-like language introduced in grammar 7.1. Since at this point the fragments are free of control flow components, this is just a syntactical adjustment. If concepts are used in the original programming language that cannot be mapped in our language or can only be realized by changing the semantics, the limits of our approach are reached. This is the case, for example, as soon as temporal considerations or pointer arithmetic are applied.

In the course of the running example, data flow information is transformed. As already mentioned, the transformation of the code fragment is limited to purely syntactical changes in the concrete case. Regarding the types used, no mappings are necessary. Besides, the control flow, which is encapsulated in method calls of the System Model, for example, is mapped by explicit `EGPPGraphs` to guarantee the requirement for atomic assignment operations in the context of `EGPPNodes`. For example, the methods `calc_permitted_speed_to_driver`, `calc_speed_to_driver`, or `calc_speed_onboard` used in the context of the statemachine from figure 16.3 (Supplementary Material) and figure 16.4 (Supplementary Material) are affected.

7.3 Architecture And Analysis Framework

Continuing the metamodeling concepts, this section deals with the implementation of an associated reference technology platform, namely the Architecture And Analysis Framework (A3F). The origins of this framework have already been described in a conference and a journal paper, which builds the basis for the contents presented throughout this section [151][148].

The A3F represents a framework that was developed especially for MDSD and can be used in this context for various purposes. To allow this flexibility in terms of purpose, the framework is built on the conceptual basis seen in figure 7.26.

The General Purpose Modeling Language (GPML) can be seen on the left side of the figure. Its model instances provide the basis for the user and at the same time the interface to the processing chain within the A3F. From artifacts of the GPMLs, different DSMLs can be derived by M2MTs in the context of the framework and vice versa. In particular, several DSMLs can be combined, which corresponds to the Omni Model approach explained in chapter 7 which are arranged in the middle of the figure. Again, building on the domain-specific representations, Purpose-Specific Data (PSD) can be created by further transformations that serve only a very specific purpose. In the following, the mentioned parts of the conceptual basis are discussed in more detail.

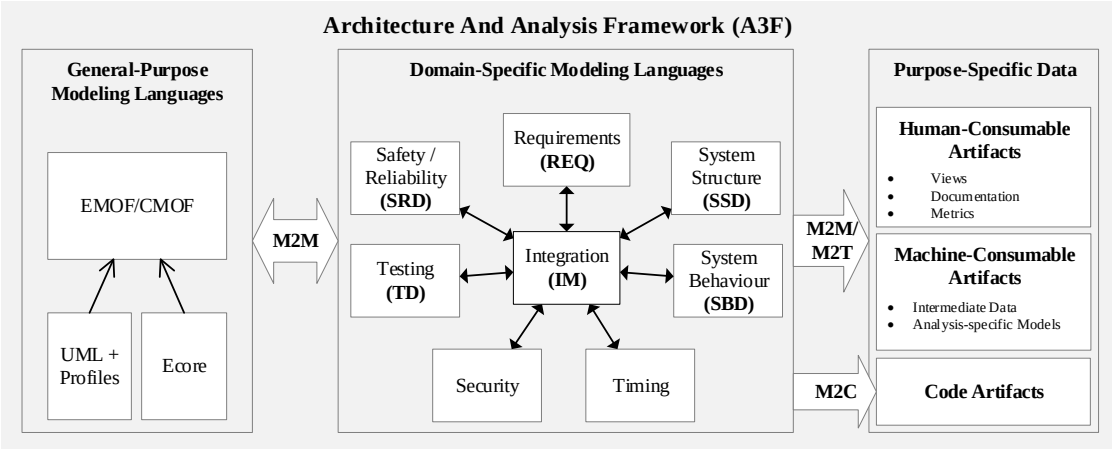


Figure 7.26: Conceptual foundations of the A3F (based on [148])

General-Purpose Modeling Languages represent the entry point into the A3F. This is chosen mainly for reasons of acceptance of the framework and in the sense of broad applicability in different development concepts. Especially in the context of these modeling languages, there is a large number of practitioners, which has resulted in a significant number of so-called MDSD tools, which enable the modeling of such languages. This variety of tools is further meant to interact with our prototype. In the context of the prototypical implementation, the GPMLs are used in two different contexts. On the one hand, they serve as a uniform basis for specifying the domain-specific model information. On the other hand, certain parts of the model landscape can be represented by a GPML-based model. Here, the user is free to choose the GPML as long as the chosen language offers concepts for metamodel extensions.

Domain-Specific Modeling Languages are the basis of the *Omni Model* approach presented in section 7.1. For this purpose, the information modeled in a GPML is converted by M2MTs into a domain-specific modeling language (if provided) and then put into relation with other models. These transformations are commonly applied in both directions, i.e. domain-specific artifacts can be transformed back to their general-purpose artifact. The number of DSMLs used depends on the development context and can be designed as required. The resulting models provide a solid basis for further processing steps, which can be used for the respective application context.

Purpose-Specific Data does not necessarily represent model data in the classical sense. Furthermore, the M2X transformations are irreversible, since a reduction concerning irrelevant information is performed during processing. On the right side, figure 7.26 enumerates possible types of PSD.

The first category is given by the *Human-Consumable Artifacts*, which mainly represent model data prepared for a special purpose and primarily for the toolchain user. There-

fore, usually M2MTs or Model-to-Text Transformations (M2TTs) are applied using established frameworks. Thus, the focus of such artifacts is not on further processing by machine. Examples of such artifacts are *Views* known from software architecture, which can be generated from the available model (landscape). Another example is the automated creation of *Documentation*. In particular, the Integrated Model Basis can produce significant added value if the focus is on the entirety of the information and its interrelationships. Finally, in the context of *Human-Consumable Artifacts*, *Metrics* can be mentioned which prepare characteristics and convert them into comparable quantified properties for the user.

In contrast, the *Machine-Consumable Artifacts* focus on machine processing. This processing can either take place in the A3F itself or can be delegated to external tools by export/import functionality. The latter generally represents the conceptual interface for connecting external tools. A special form is the *Analysis-Specific Models*, whose representative EGPP has already been presented in detail.

Code Artifacts are mentioned as the last representative of PSD. Depending on the modeling languages and the development methodology applied, the Integrated Model Basis provides a more or less optimal starting point for generating target code. This can be achieved by transformations as well, in this case, M2CTs, which is out of scope for this thesis.

7.3.1 Framework Architecture

Based on the conceptual view on the framework, a prototypical implementation of the A3F is realized. To meet different application purposes and technological contexts and to reflect the extensibility and flexibility of the concept, the following architecture is chosen (figure 7.27).

GPML artifacts represent the point of interaction with the framework regarding data. Therefore, all information that is modeled in *Enterprise Architect*®, a CASE tool that was chosen during the prototypical implementation, is stored in a connected model repository. Furthermore, this repository is the source of information for the A3F developer. In the following, individual components of this toolchain and in particular, the internals of the technical realization of A3F is described in detail.

Model Repository

First, the *Model Repository* is realized by a database. For instance, it could be a relational Postgres database, not necessarily running on the same hardware as the framework or the modeling tool. The applied schema of the relational database must be able to map and manage models according to any GPML including extension mechanisms. A solution satisfying these requirements is given by Enterprise Architect®. Therefore, in the context of our prototype, the predefined database schema from Enterprise Architect® is

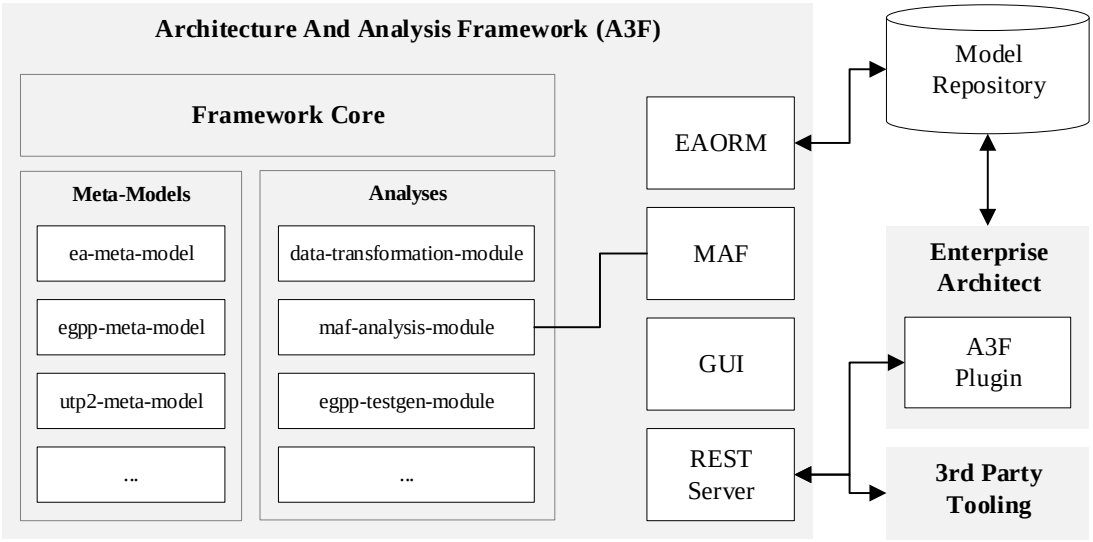


Figure 7.27: Internal architecture of the A3F

applied, which makes the interaction between the repository and the CASE tool work out of the box.

In contrast, A3F must provide a way to work with the information in the *Model Repository*. A special Object Relational Mapper (ORM) has been developed for this purpose (see section 7.3.1). All in all, the architecture decision regarding a single and fully-fledged model repository provides a solid basis for flexible, distributed, and scalable work with the A3F.

Enterprise Architect®

As already mentioned, a special CASE tool for creating the GPMLs models was chosen in the course of the prototypical implementation, namely Enterprise Architect®. Enterprise Architect® is one of many modeling tools that are widely used in industry. Especially the high distribution among industry partners in the context of the research projects mentioned at the beginning of this thesis has led to the choice of this tool. However, Enterprise Architect® has several technological unique selling points that have additionally supported the choice.

On the one hand, the extension possibilities regarding the modeling support of DSLs are to be emphasized. This makes the use of the MDSD tool for modeling the DSLs explained in the concept more user-friendly, which has been implemented for some modeling languages in this thesis. On the other hand, Enterprise Architect® offers the possibility to extend the standard range of functions with plugins or to adapt it to your needs. This mechanism is used within the scope of the prototypical implementation.

In addition to the wide distribution and the mentioned extension possibilities, there are some technological aspects to mention, which are beneficial to the technical implementation of our prototype.

- Integrated baseline mechanism: a snapshot of model elements and a possibility to restore a baseline
- Transparent data model: Extensive documentation on the repository data model available
- Scalable data management: support of various database systems/technologies

All in all, the Enterprise Architect® thus provides a solid basis within the course of our prototypical implementation. However, the framework architecture enables the adoption of company-specific tooling, replacing Enterprise Architect® with any other CASE tool.

The Framework

On the left side of figure 7.27, the functional parts of the framework are shown. Here, a further distinction is made between the metamodel definitions, the analysis definitions, and the framework core.

A3F Metamodels At this point a multitude of metamodels including the corresponding transformation rules are defined. These can be classified according to the three-part division is shown in figure 7.26. In the first category GPMLs, the `ea-meta-model` is to be mentioned as a representative, whose components are explained in part VI (Supplementary Material). It is an Ecore representation of the Enterprise Architect® data model, which is the A3F counterpart of the *Model Repository*. In particular, the *EAORM* uses data from the Model Repository to create instances of this metamodel, which are then suitably processed. Work may either be carried out directly on the created models or a specified M2MT is applied to obtain a domain-specific representation.

In the category of DSLs, there is a variety of representatives, indicated by the dots in the figure. In contrast to the metamodels described above, instances of the `u2tp-meta-model` (see section 7.1.2), for example, are always derived from GPML instances. The combination of such metamodels together with the Integration Model represents the Omni Model approach in the context of the A3F.

The last remaining category is described by metamodels, which can be assigned to the presented PSD. The `egpp-meta-model` (section 7.2.1) should be mentioned here as a representative *Analysis-specific Metamodel*. In principle, its instances can be derived from both of the presented metamodel categories by M2MTs, but usually, a domain-specific metamodel is used as the source due to the more concrete semantics.

A3F Analyses At this point, the processing modules of the framework are integrated, which ultimately specify the scope of functions. Here, a module can specify any number of so-called *analyses*, which represent the smallest processing units. For example, the loading (`ea_db_loader`) and saving (`ea_db_persister`) of models from the *Model Repository* are implemented by separate *analyses*. In addition to the specification of processing logic, which in turn requires certain metamodels from the previous section, other frameworks can be integrated. Specifically, the module *maf-analysis-module* integrates the Model Analysis Framework (MAF), which was developed at the professorship. The following list shows a selection of the current A3F analyses:

- `history_loader`: Load specific analysis results from previous executions
- `data_transformer`: Runs a model-to-model transformation script
- `fta_cutsets`: Computes cutsets for a fault tree
- `documentation_generator`: Generates a set of hypertext documentation for a set of model artifacts

Since a certain amount of configuration is necessary for addition to the pure implementation of the functional logic, the following DSL was developed (see listing 7.5), which allows the specification of these configuration parameters and allows for a combination of analyses.

```
1  configuration:
2    (analyses+=analysis)* ;
3
4  analysis:
5    'analysis' classID=ID '('identifier=ID')'
6    (('{' (parameters+=parameter)* '}') | ';') ;
7
8  parameter:
9    id=ID '=' value=STRING ';' ;
10
11 ID : LETTER (LETTER|DIGIT)* ;
12
13 fragment LETTER : [a-zA-Z\u0080-\u00FF_] ;
14 fragment DIGIT : [0-9] ;
15
16 STRING: '"' (~('"' | '\\' | '\r' | '\n') | '\\' ('"' | '\\'))* '"' ;
17
18 BLOCK_COMMENT: '/*' .*? '*/' -> channel(HIDDEN) ;
19
20 LINE_COMMENT: '//' ~[\r\n]* -> channel(HIDDEN) ;
21
22 WS : [ \t\r\n]+ -> skip ; // skip spaces, tabs, newlines
```

Listing 7.5: ANTLR grammar for A3F configuration language

As the ANTLR grammar shows, a so-called *configuration* consists of any number of analyses. Each analysis is specified by a unique *classID* and an instance *identifier*. This makes it possible to use several instances of the same analysis in one configuration. Besides, a set of parameters can be specified for each analysis in the form of key-value pairs.

Dependencies between individual analyses can be created in different ways. There are data dependencies that result from interdependent parameters as well as there are implementation-related dependencies that result from splitting processing steps across several analyses.

A3F Core This part of the architecture forms the framework around the components already explained. In particular, *Core* implements the execution of an analysis configuration. For this purpose, the transferred configuration is first checked for certain properties that determine its validity:

- Syntactic conformance to the defined configuration grammar
- Absence of circular dependencies between included analyses
- Proper specification of mandatory analysis parameters
- Presence of dependencies (data and implementation) across individual analyses

If the configuration was classified as valid, an execution sequence is determined from the underlying dependency graph. This sequence is chosen as optimally as possible by running independent analyses in parallel. In addition to the pure determination of the execution sequence, the administration of the processing results of the individual analyses and their exchange is managed. This means the framework manages all results of analyses in so-called *ResultContainers*. This applies to each execution round of a configuration, as well as across successive rounds. The presented components already allow functional variants of the A3F, with the possibility for customizing functional scope.

Framework Extensions

The extensions improve the interaction with the framework. To extend the scope of functions and interaction with other tools, additional components are planned, explained at the end of this section.

Developer GUI - This A3F interface allows users to elegantly create and edit configurations, checking their well-formedness at all times. Furthermore, the execution can be controlled, and especially the results of an analysis run can be examined in different views, e.g. for detailed examination of model transformation results as well as for visual preparation of model artifacts. In addition to the presentation of analysis results, the user is provided with detailed log information on the execution of a configuration. The main view of the GUI is shown in figure 7.28.

Server - To scale the framework better and to create a versatile interface, a server component was developed. This encapsulates the already presented framework and

provides all interaction points in form of a REST-API. All model data can be retrieved via this interface and thus be processed in other tools. Furthermore, the server component marks the basis for other extensions, which are explained in the following.

Enterprise Architect® Plugin - As already mentioned, the Enterprise Architect® used in the prototypical implementation offers the possibility to extend its functionality with plug-ins. In this way, a further type of interaction with A3F was implemented. The plugin enables analysis configurations to be executed directly from Enterprise Architect® and allows the calculated results to be visualized directly. The REST-API mentioned before is used here, which guarantees the decoupling of the CASE tool and the framework.

Future Extensions - In principle, further useful extensions are conceivable, but these were not realized in the course of the prototypical implementation. First of all, the realization of a frontend should be mentioned here, which is based on the REST-API and thus represents a more practicable implementation compared to the *Developer GUI*. In particular, this allows for better scaling and cooperative work with the framework. Besides the user interface, the data interface offers great potential for extensions. At this point, a connection of the framework to other model data sinks is useful and necessary to be able to leave the context of the MDSD tool. Conceivable is the connection to the so-called *ModelBus*, which is a technology that connects different tools in the context of MDSD and enables data exchange [35]. A technology that is used in the *ModelBus* and thus represents a product-independent extension of A3F's functional scope would be a connection according to the *Open Services for Life Cycle Collaboration (OSLC)* [132]. This enables the connection to other tools that implement an OSLC interface.

7.3.2 Working with the Framework

Based on the remarks on the concepts and architecture of the A3F, the following section briefly describes how to work with the framework. The already mentioned *Developer GUI* forms the basis for this (figure 7.28).

Besides, the already known CSM Omni Model is used to illustrate common steps. The application scenario is the loading, transformation, and linking of all model artifacts that participate in the Integrated Model Basis. The configuration representing this functionality is developed on the left side of the Developer GUI using the *Analyses Repository* and the textual configuration editor. The configuration in listing 7.6 represents the result of this work.

```
analysis ea_db_loader(system) {
  dbkey="r-db3";
  requests="EADiagram|{21BECB65-204D-4257-B8F3-8131A7CE9C50}";
}
analysis data_transformer(system) {
  request="EnterpriseArchitectPackage|SysmlPackage";
  inputs="ea_db_loader|system|DataTransformationResult|getOutputElements";
}
analysis ea_db_loader(test) {
  dbkey="r-db3";
  requests="EADiagram|{C50814AB-C0B5-41ca-8BF9-E17676D937E1}";
}
analysis data_transformer(test) {
  request="EnterpriseArchitectPackage|U2tpPackage";
  inputs="ea_db_loader|test|DataTransformationResult|getOutputElements";
}
analysis ea_db_loader(integration) {
  dbkey="r-db3";
  requests="EADiagram|{1EB38DDF-663D-4ae7-9C39-A605DB898BEE}";
}
analysis data_transformer(integration) {
  request="EnterpriseArchitectPackage|IntegrationModelPackage";
  inputs="ea_db_loader|integration|DataTransformationResult|getOutputElements";
}
analysis data_transformer(systemegpp) {
  request="SysmlPackage|EGPPPackage";
  inputs="data_transformer|system|DataTransformationResult|getOutputElements";
  ppParameters="codeprocessor|c";
}
analysis data_transformer(testegpp) {
  request="U2tpPackage|EGPPPackage";
  inputs="data_transformer|test|DataTransformationResult|getOutputElements";
}
analysis im_validation(imvali) {
  validationconfig="11111";
  targetIMs="{1EB38DDF-663D-4ae7-9C39-A605DB898BEE}";
  inputs="data_transformer|integration|DataTransformationResult,
    data_transformer|testegpp|DataTransformationResult,
    data_transformer|systemegpp|DataTransformationResult,
    data_transformer|test|DataTransformationResult,
    data_transformer|system|DataTransformationResult";
}
```

Listing 7.6: Example configuration for A3F based on the Running Example

As already indicated, the analyses specify different sets of parameters controlling the internal processing of the model data. What can be seen is, that the majority of analyses are based on each other, with some analyses specifying parameters that in turn use the results of other analyses as input parameters, e.g. `inputs="ea_db_loader..."`. This results in data dependencies. The dependency graph determines a valid execution order. This graph can be seen in the upper left corner of the Developer GUI, while the nodes represent the participating analyses and the edge determine a dependency between the aforementioned analyses.

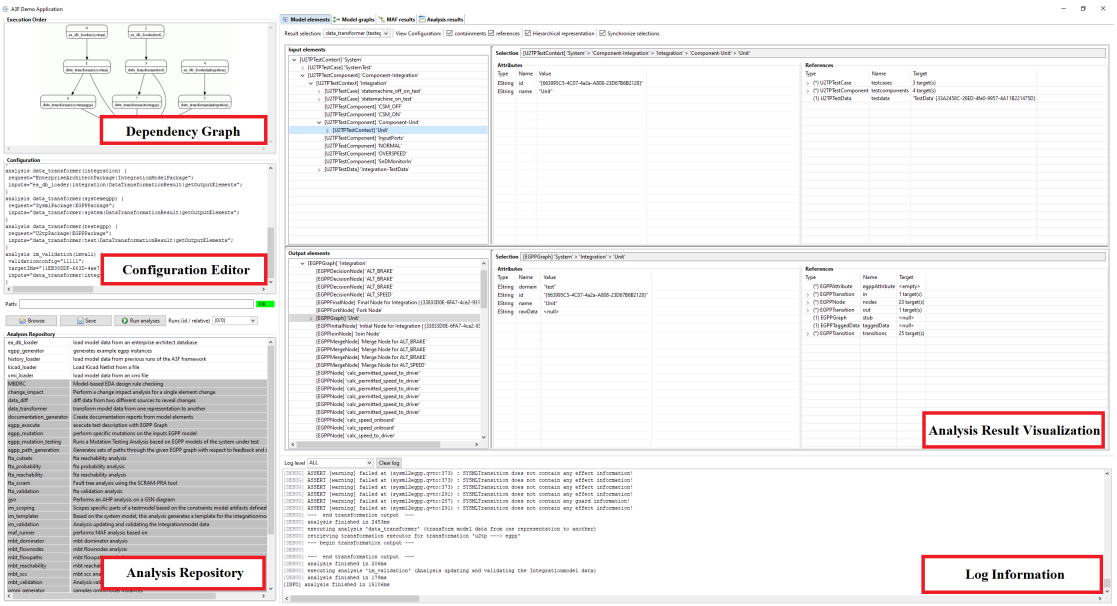


Figure 7.28: Main view of the A3F Developer GUI

Once the configuration has been checked and validated, it can be executed by the framework by pressing the *Run Analyses* button. After execution is completed, the different views for analyzing the processed results are available as tabs in the main view. In our case, the processed data of the analyses `ea_db_loader` and `data_transformer` can be examined mainly through the *Model Elements* view and the *Analysis Results* view.

Finally, at the bottom of the graph, the log area is shown, which, when executing the configuration shown, merely provides information on which processing steps were carried out and in what time frame this was accomplished. In other cases, this area can provide information about possible causes of errors. In the following chapters, which cover the sub-steps of the MCSTLC, a supplementary part to the configuration shown above is detailed in each case, which illustrates the concepts in the context of the A3F.

7.4 Related Work

To better understand the Omni Model approach presented, this section discusses related approaches and shows the differences between them and our approach. However, since the concepts of the Omni Model approach and the related prototypical implementation are multi-layered and no comprehensively comparable approach is known to us, this section is subdivided by the core aspects of our approach. In particular, the aspects of *Model Integration* and *Analysis-specific Metamodels* are examined more closely.

Model Integration

The literature presents various approaches to the integration of development information/models. A classical method for linking information from different development artifacts is the so-called requirements tracing. Finkelstein et al. elaborated on the underlying traceability problem, which represents the major challenge to establish forward and backward traceability between requirements and target code [79]. Technically, a requirement management tool such as *IBM DOORS®* is usually used for the realization. The Omni Model approach, in contrast, provides more than just the pure traceability of requirements through the Integration Model.

In addition to the implicit coupling of structure and functionality via requirements, independently implemented aspects of the system, in terms of *Aspect-Oriented Modeling/Design*, can be integrated using *Model Weaving*. The identification of suitable integration points and the subsequent integration of the additional aspects plays a central role [99]. Alternatively, model weaving is applied on metamodel-level to describe relations between different models by a so-called weaving model, which subsequently enables integration of model information by transformations [104][50]. However, such techniques are usually strongly focused on functional aspects and aim to create a new combined model artifact. In contrast, the Omni Model approach maps both the structure and the behavior, which opens up the same possibilities. However, the strict separation of the concerns is maintained, as there is no intention to create an all-encompassing model artifact.

Especially in the context of Model-Driven Engineering (MDE), the integration of model information is achieved by the so-called *Mega-/Macromodeling*. Here, relations between entire models/metamodels are established by a separate model artifact, whereby the content of the linked models has no relevance for the linkage [86][34]. This type of combination of model artifacts can be combined with the previously mentioned technique of model weaving. Bezivin et al. have presented an approach that enables the coordination between different artifacts using the aforementioned techniques [105]. This is comparable to the Omni Model approach, which is more concrete in terms of the semantics of the connections between the modeling domains. This allows us to make certain assumptions regarding the integrated view on parts of the Integrated Model Basis in later processing steps, e.g. the Abstract Test Case Execution.

Approaches that manage the integration of model data via a uniform information base usually do not offer flexibility concerning the specification languages used. Furthermore, the aspect Separation of Concerns is often implemented via view/viewpoint-like constructs, which is sufficient in many cases [92]. In the automotive context, EAST-ADL deserves to be mentioned here, which, in addition to vertical differentiation about different abstraction levels, provides for horizontal differentiation of concerns such as dependability or variability.

Analysis-specific Metamodels

There are several related approaches in the area of specific model artifacts for analysis. However, most approaches are tailored to very specific problems and therefore cannot cover all facets of our approach satisfactorily. Especially if the focus is on the execution of model artifacts, there are some comparable approaches. In the context of UML, there is the Foundational Subset for Executable UML Models (fUML), which deals with the executability of UML models and the corresponding semantics of the models concerned [70]. In conjunction with the Action Language for Foundational UML (ALF) which allows a textual description of fUML concepts, executable models can be specified [17]. It would be conceivable to specify the model artifacts of the Integrated Model Basis according to fUML or to translate them via model transformations into an fUML-compliant variant. However, on the one hand, this restricts the circle of users considerably and on the other hand, this solution is rather designed for formal checks, which does not reflect the focus of our approach. In the context of our approach, missing/spurious semantics of the original artifact can be enriched so that the characteristics of flexibility and early applicability are preserved.

Another possibility is given by the statecharts developed by David Harel [84]. The formal syntax and semantics of these statecharts allow checking the correctness of the modeling even before the first line of code exists. The strictly formal idea behind this type of specification, however, brings the same problems with it as were mentioned in the context of fUML. Another problem is the missing description of the structural hierarchies of the model under consideration. Especially in the context of the creation of specific model sections (see chapter 8) this leads to problems, but this is explicitly covered by our implementation in the form of EGPP.

In addition to the modeling possibilities mentioned above, there are many other possibilities, such as different variations of Petri nets [31]. However, variants for analysis are often used here that is strongly tailored to the application context and are therefore difficult to use for other purposes. Furthermore, in many cases, insufficient documentation and the availability of a prototype implementation questions the practical applicability.

7.5 Conclusions and Outlook

In the context of this chapter, we have presented a comprehensive approach that combines models of different development domains/disciplines and prepares them for analysis purposes. In particular, we focused on necessary test activities during model-based development. The question formulated in section 1.1

What kind of modeling conventions and metamodeling concepts are essential to effectively apply testing activities in the early stages of MDSD?

can therefore be answered as follows. The presented Omni Model approach allows using the modeling languages commonly utilized by the user, provided that the minimum requirements for the subsequent integration are met. These are realized in the respective domains utilizing a set of modeling guidelines and thus form the basis for the modeling landscape. A model artifact created explicitly for integration purposes provides functionality for both structural and behavioral aspects and encapsulates information important for model-centric testing. By defining a special metamodel for the analysis-specific view of the model landscape, i.e. EGPP, the basis for uniform processing of all model information is established. This representation is automatically derived from the original model artifacts through M2MTs. In particular, a sound basis was created for all processing steps of the MCSTLC. In addition to the conceptual solution of the problem definition, a prototype implementation was developed. The Architecture And Analysis Framework (A3F) created for this purpose offers extensive possibilities for the processing and analysis of model information, whereby the central concept of the framework is again the Omni Model approach. The focus was on the practical applicability of the implementation as well as the integration into existing toolchains.

Each concept has certain strengths and weaknesses, which have to be communicated to the user such that a well-founded decision for or against the application of the approach can be made. For this reason, the following section deals with the advantages and disadvantages in the form of a brief discussion. Among the advantages of the current approach is the adaptability to the quantity and characteristics of the included model artifacts. In this context, the strict separation of concerns is advantageous, since no undesired cross-relationships and emergent behavior arise. Furthermore, the simple and intuitive display of cross-relationships and the specification of meta-information is another positive aspect. On the side of the technical implementation, the existence of a prototype implementation is to be emphasized, which shows the practical applicability. Especially the applicability from very early phases up to late phases of model-centric developments gives the user an advantage.

Addressing the approach's shortcomings, the mentioned functionalities are based to a large extent on a new and additional model artifact introduced especially for this purpose, namely the Integration Model. At this point, the additional effort of model creation and maintenance has to be taken into account. However, this effort can be kept relatively low by sophisticated support from the tools and strict implementation of modeling guidelines. Also, the effort for the specification of the M2MTs for the automated derivation of the analysis-specific representation should not be neglected. Besides, this artifact has to be created by experts, since a high degree of modeling experience and domain knowledge is required. Conceptual errors in the implementation of these transformations have serious effects on all downstream processing steps. There are some shortcomings on the technical level in the current implementation. These primarily concern the usability of the implementation and the connection to more extensive and flexible interfaces for model data, such as OSLC or the *ModelBus*.

In conclusion, possible starting points for future improvements/extensions on the side of the concept as well as the connected prototypical implementation are shown. On the conceptual side, the current integration of data flow information via the internal

pseudocode-like language is to be mentioned. In later stages of expansion, further concepts typical for certain programming languages can be taken into account to increase the functional extent. Likewise, the connection of the concepts just mentioned for the exchange of model information between tools (OSLC, ModelBus) represents a possible future extension of the implementation [132][35]. The same applies to the improvement of the user interface to enable the most intuitive interaction possible.

8

Model-Based Test Case Management

Based on the Omni Model of the SUT, whose basic concepts are explained in the previous chapter, the first processing step of the MCSTLC takes place. In the context of this step, the test focus and the test level are mapped to the model landscape and thus lay the foundation for subsequent processing, such as test case generation. As already described at the beginning of chapter 6, some phases of the classical STLC are addressed at this point, albeit in a different order. In particular, the concepts presented in the following allow for the selection, prioritization, and reduction of test cases in an early phase of the MCSTLC before test cases are even derived from the model.

In contrast to the largely manual modeling work on the Integrated Model Basis, this process step represents the first part of the automated processing chain. In early iterations of the MCSTLC, i.e. at low integration levels, only the basic models created by the modeler/developer are processed. In contrast, in later iterations of this processing step, i.e. on higher integration levels, findings of the downstream test execution are fed back into the model and thus taken into account.

As the excerpt of the MCSTLC in figure 8.1 shows, besides the already mentioned Integrated Model Basis, some configuration parameters are necessary to realize the mentioned functionality.

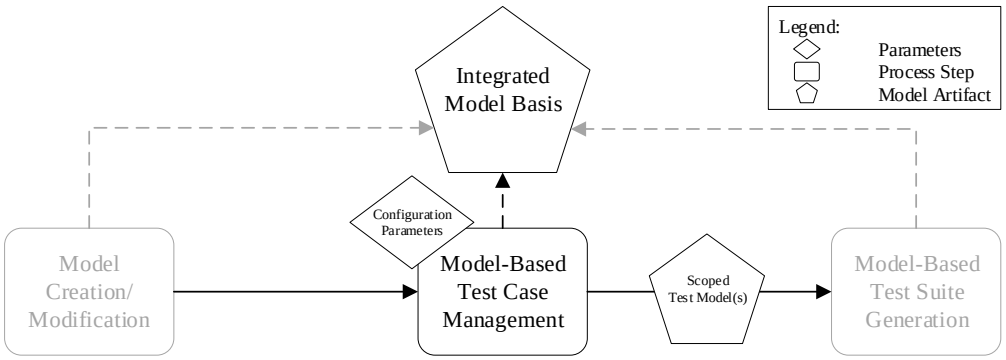


Figure 8.1: MCSTLC extract focusing the Test Case Management and involved information

The test engineer's test focus must be able to be specified intuitively, again utilizing and expanding the Aspects concept (see section 7.1.3). The test level (Unit, Integration,

System) has to be applied to the set of models, which correspond to the specified test focus. For this purpose, the Execution Graph++ Metamodel (EGPPMM) was introduced. Based on this set of information, the processing can be performed to produce one or more so-called *Scoped Test Models*, each of which represents very specific portions of the original Test Model. Therefore, the algorithmic solution that realizes the extraction of these sub-models is called *Test Model Scoping*.

8.1 Prerequisites for Test Model Scoping

To provide the necessary foundations for algorithmic processing in the context of Test Model Scoping, the relevant concepts are introduced in this section. In addition to the obvious configuration parameters like the test level, the mechanism for the specification of the test focus is presented as well as the relevant parts of the Omni Model.

8.1.1 Test Focus Specification

As previously mentioned, the extraction of a specific Test Model is the overall goal of this process step. *Specific* in this context means that the resulting model artifact and test cases tackle the focused System Model parts. Consequently, the generated test cases can challenge the focused parts as efficiently and effectively as possible. In this case, the Aspects concept introduced in section 7.1.3 serves as a basis, which realizes a uniform interface for intrinsic and synthetic information of the Integrated Model Basis.

Aspects Revisited - The Aspect Constraint Language

The *Aspect Definition Language* allows specifying the available Aspects and their value ranges. Furthermore, the concrete values for model elements of the Integration Model, which reflects the instantiated version of the SUT in terms of its structure, are retrieved or specified using the *Aspect Specification Language*. Similar to RBT, where risk assessments are used as a basis for the selection of test cases, this concept allows any information of the Omni Model to be included in the process [18][65].

To describe the test focus and to extract the model parts accordingly, the *Aspect Constraint Language* is developed. The language enables the user to specify constraints for certain Aspects and to combine them with logical operators. Listing 8.1 shows the grammar that realizes the mentioned functionalities.

In combination with the *Aspect Specification Language*, the name of the Aspect serves as a reference to the respective definition. Starting from the Aspect name, the elementary constraints are described. Depending on the type of the underlying Aspect (ranged vs. set), different operators can be used to specify the constraint.

```
acdsl: term;

term:
  factor (boolbinaryop factor)*;

factor:
  constraint | boolunaryop factor | '(' term ')';

constraint:
  cname=ID ':' coparator=OP '[' cdefpart ']' (':' clogicop=LOGOP )?;

cdefpart:
  ( value=ANYID ',')* value=ANYID;

boolbinaryop: BOOLAND | BOOLOR | BOOLXOR;

boolunaryop: BOOLNOT;

BOOLAND: '&';
BOOLOR: '|';
BOOLXOR: '^';
BOOLNOT: '!';

LOGOP : 'and' | 'or' | 'xor';
OP : 'le' | 'lee' | 'in' | 'gre' | 'gr';

fragment LETTER : [a-zA-Z\u0080-\u00FF_];
fragment DIGIT : [0-9];

ANYID : '\'' ( LETTER | DIGIT | '.')+ '\'';
ID : (LETTER(LETTER|DIGIT)* | NUMBER);

NUMBER : '-'? ('.' DIGIT+ | DIGIT+ ('.' DIGIT*)? );

BLOCK_COMMENT: '/*' .*? '*/' -> channel(HIDDEN);

LINE_COMMENT: '//' ~[\r\n]* -> channel(HIDDEN);

WS : [ \t\r\n]+ -> skip ; // skip spaces, tabs, newlines
```

Listing 8.1: ANTLR grammar for the Aspect Constraint Language

For Aspects, which define a continuous value range, a truth value depending on the threshold value (`cdefpart`) can be derived using comparison operators (`coperator`: < (`le`), <= (`lee`), = (`in`), >= (`gre`), > (`gre`)).

In contrast, for Aspects with a discrete set of values, only the overloaded `in` operator can be used, which checks each concrete value of the Aspect, whether the respective element is contained in the set (`cdefpart`). For each pairwise comparison, a truth value is derived, which contributes to the overall result. How these partial results are linked to each other can be specified by three logical operators (`clogicop`: `and`, `or`, `xor`).

The elementary constraints can be combined into complex constraints. The unary logical negation operator (`!`) and the binary operators `AND` (`&`), `OR` (`|`), and `XOR` (`^`) are

applicable. Beyond, parenthesis is supported for clarity and binding.

All in all, the presented language offers the necessary constructs to represent the test focus of the test engineer. Therefore, the user has to develop an *Aspect Constraint Language* expression made available to the automated processing chain. An exemplary application of the *Aspect Constraint Language* can be seen in the course of the introduction of Test Model Scoping in section 8.2.

Execution Graph++-based Test Levels

Besides the constraint-based extraction of a specific part of the Integration Model, the classic test levels are mapped in the context of the EGPP-based representation of the original Test Models. Figure 7.24 in section 7.2.2 shows the concepts behind the EGPP-based Test Models. In particular, the connection to the stub concept is shown, which plays an important role in the further course.

Essentially, a system-level Test Model is defined by the fact that it is not itself integrated into another Test Model but integrates other Test Models. A unit-level Test Model, on the other hand, is integrated into other Test Models but does not itself include any Test Models. Any Test Models that include other Test Models and are themselves included in another Test Model are called integration-level Test Models. Further, there are no restrictions on the number of integration levels in the Test Models.

Analogous to the specification of an Aspect Constraint Language expression, the test engineer must specify the desired test level and make it available to the automated processing chain as a configuration parameter.

8.1.2 Excerpt of the Omni Model

The Omni Model, or Integrated Model Basis, containing the Test Model and its relations to the Integration Model covers the structural relations and relations regarding the specified control flow. All these relations are specified on the original model and are further incorporated into the internal EGPP representation within the context of the M2MTs presented in section 7.2.2.

Furthermore, in the context of Test Model Scoping, the aspects defined and specified in the Omni Model play an important role and have a significant influence on the quality of the scoping results. The quality of these results can be improved by either connecting additional model information via the Integration Model, or by maintaining custom Aspect information in the Integration Model for a more precise focus specification.

8.2 Test Model Scoping

The term Test Case Management conceals different goals concerning the resulting set of test cases for a certain test objective. Specifically, the disciplines selection, prioritization, and reduction are covered by a corresponding configuration of the same algorithmic solution. These three disciplines pursue a similar goal, namely the selection of a subset of test cases, where the contained test cases fulfill a certain objective. Projected to the model-level, the aim is to extract sub-models from the entirety of the Test Models. The term Test Model Scoping (TMS) is used as a representative term for the algorithmic solution.

Based on the Integrated Model Basis and the configuration parameters explained in the previous section, the TMS process is divided into three phases. The first step *Integration Model Based Filtering* reduces the original Integration Model based on the Aspect Constraints. Based on the results, the subsequent step *Test Model Mapping and Reconstruction* exploits the cross-domain connections between model artifacts to derive Test Model artifacts that are related to the extracted Integration Model parts. However, the resulting set of model elements of the Test Model does not necessarily represent a valid Test Model in the sense of a valid input for the subsequent test case generation. Therefore, in the same processing step, a valid Test Model is reconstructed from the set of Test Model elements. The last step of the processing chain *Test Model Split and Enrichment* deals with the split of the derived intermediate Test Model alongside integration levels to end up with a set of Scoped Test Models of uniform integration level. Further, the enrichment of these Scoped Test Models depending on the choice of the subsequent test case generator represents a task of this process step. Altogether, this step of the MC-STLC represents a target-oriented preparation of the model basis for the downstream processing steps.

The contents of the following sections on Test Case Management and the TMS process represent a continuation of the concepts of the conference paper [145].

8.2.1 Integration Model based Filtering

As already mentioned, the fully-blown Integration Model marks the starting point of the TMS process and thus for the *Integration Model based Filtering* step. The goal of this processing step is to extract a submodel that corresponds to the focus of the test engineer and serves as a starting point for further processing steps. The Aspect Constraint Language introduced in section 8.1 allows the test engineer to specify the test focus. The evaluation of this Aspect Constraint expression in turn is based on the concrete Aspect Specifications of the model elements of the Integration Model. The affected elements (`IMTreeNode`) of the Integration Model form a tree structure, which is schematically represented in figure 8.2 and is used for further explanations. Furthermore, the dashed connectors represent the type `IMGeneralization`, and the solid connectors the type `IMPartOf`.

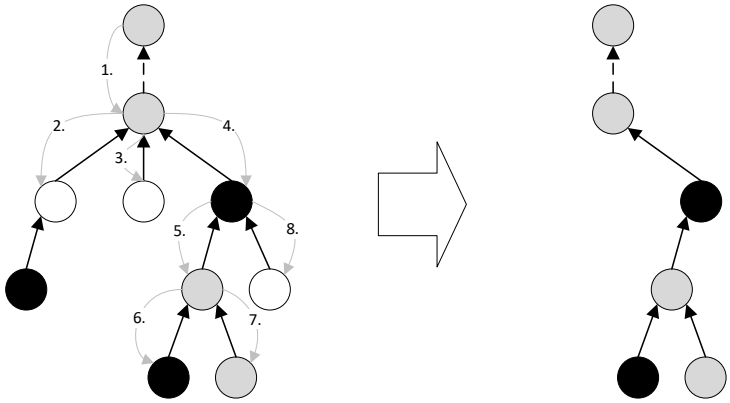


Figure 8.2: Algorithmic approach for Integration Model based filtering

The different colors of the nodes represent the result of the evaluation of the Aspect Constraint Snippet on the respective Aspect Specification. A black node means that the evaluation was positive, i.e. all analyzable constraints and their combinations are fulfilled. A white node, however, symbolizes a negative result of the evaluation. A gray node means that no evaluation can be performed since the node does not specify any information about the respective Aspects. This case is treated in the same way as a positive evaluation about the resulting set of model elements of the Integration Model.

The evaluation of the Aspect Specification Snippets starts at the root node of the tree structure. However, if only a subtree of the original Integration Model is to be considered, the test engineer has the additional option of setting a different node as the starting point for the evaluation. In case of a positive evaluation of the root node's Aspect Specification, the evaluation for the child nodes is triggered. If the result of the evaluation is negative, the top-down process for the respective branch of the tree structure ends, and all nodes below are evaluated as negative. A special case comes into effect as soon as two nodes of the tree structure are connected with an `IMGeneralization` (see figure 8.2 top left). In this case, if the higher-level node has been evaluated positive, the node below is evaluated positively, although the evaluation was not yet performed.

As soon as this process stalls, the determination of the reduced Integration Model is finished. The right side of figure 8.2 shows the result schematically. Furthermore, if the combination of Aspect Constraints is too restrictive, the resulting set of model elements remains empty. In this case, it is either necessary to reshape the formulation of the Aspect Constraint Snippet or gives an indication to rework the related model parts. To illustrate the presented functionality, the running example from section 6.2 is utilized. In the context of the Integration Model explanations (see section 7.1.3) some Aspect Definitions have already been defined, which are listed below:

```
partID:String:set ['CSM','RSM','TSM'];
    safetyLVL:Integer:ranged [0,5];
     devPRIO:Integer:ranged [0,10];
```

Based on this, the model elements of the tree structure specify concrete values for the respective Aspects. However, it is legitimate that not every model element specifies a value for every Aspect. For the already introduced Integration Model of the Running Example, the values of the Aspects embedded in figure 8.3 are specified. The determination of these concrete values may have various origins. For example, the `safetyLVL` can be an abstract representation of a Key Performance Indicator (KPI) of a Failure Modes And Effects Analysis (FMEA) of the respective system, whereas the `devPRIO` may represent an abstract quantity that prioritizes components based on planning documents.

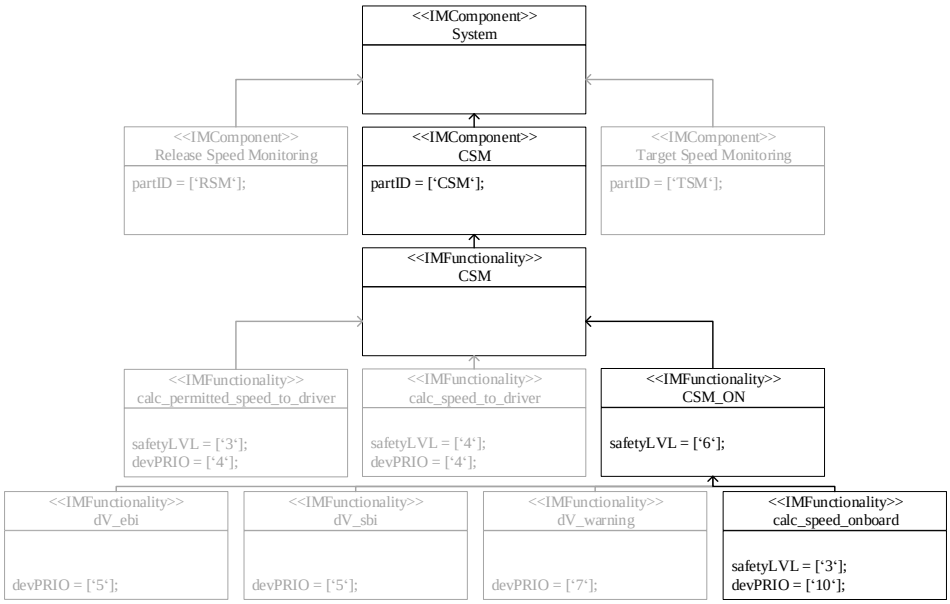


Figure 8.3: Algorithmic approach *Integration Model based Filtering* applied to the Running Example

Based on this starting point, the test engineer defines the test focus as follows:

```
partID:in [’CSM’] & safetyLVL:gre [’3’] & devPRIO:gr [’7’]
```

In concrete terms, the focus is put on components that can be assigned to the Ceiling Speed Monitoring Subsystem, which is categorized greater than or equal to 3 in terms of safety level and has been assigned a priority value greater than 7 by the development team. For any evaluated model element i.e. all the mentioned constraints need to be fulfilled (`AND` operator) to satisfy the test focus. Evaluating the tree structure presented in figure 8.3 accordingly, the grayed-out model elements are excluded from downstream processing. Overall, this can significantly reduce the tree structure, which in turn constitutes the starting point for the next processing step of TMS.

8.2.2 Test Model Mapping and Reconstruction

Starting from the resulting model of the *Integration Model based Filtering* process step, *Test Model Mapping and Reconstruction* is carried out. In this step, the mapping relations between the Integration Model and the EGPP representation of the Test Model specified by the modeler are utilized. As already described in the modeling guidelines for the Integration Model (see section 7.1.3), the balance between a very fine granular modeling of mappings and a very loose specification of these relations has to be determined at this point. The specification of as many cross-relations as possible would be advantageous for the results of this processing step, but in turn, has rather negative effects on the explorative possibilities of the subsequent Abstract Test Case Execution (see chapter 10). Starting with details of the solution, as illustrated in figure 8.4.

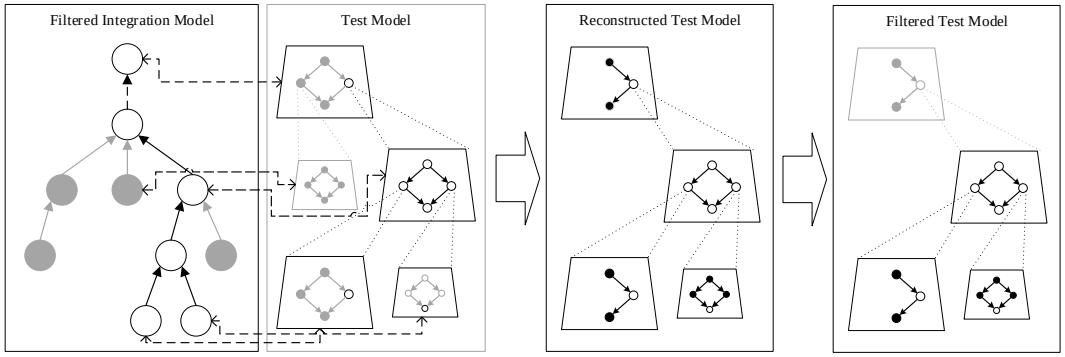


Figure 8.4: Algorithmic approach for Test Model Mapping and Reconstruction

On the left side of the figure, the schematic Integration Model of the last section is included. The grayed-out nodes and edges represent the model elements, which were filtered during the *Integration Model based Filtering* process step. Nevertheless, the fully-blown Integration Model still contains mapping information (dashed horizontal connectors) for all types of nodes, whether filtered or not. Therefore, in the first step, this mapping information is used to project the filter status of Integration Model elements onto the connected Test Model elements. Thereby, the mapping information can be utilized on different types of model information. Structure-giving components can be mapped, including the encapsulated behavioral descriptions and mapping can be specified on this basis, whereby the behavioral descriptions represent a fragmented version of the original model artifact.

The result of this projection is represented by partially grayed-out components analogous to the ones of the Filtered Integration Model. In the middle of the Test Model area, a completely mapped sub-model as well as an excluded sub-model can be seen. In contrast, the sub-models at the lower edge of this area are only partially mapped. These partial Test Models do not represent valid models instances in the sense of the EGPP metamodel definition.

To convert the set of mapped Test Model components back into a valid EGPP representation, it is necessary to perform a reconstruction step. The reconstruction concerns especially the control flow between the `EGPPInitialNodes` and the `EGPPFinalNodes` of

an EGPPGraph. Due to the possibility of selective mapping of elements of the behavioral description (elements of the aforementioned control flow), the path to a mapped element may be corrupted. If such defects in the Test Model or its sub-models are not corrected, test cases would be erroneously excluded during the subsequent Test Case Generation (see chapter 9).

Therefore, a static control flow analysis is carried out to restore the corrupted paths. In order to be able to implement this on the control flow-like representation of the EGPP models, the concepts underlying the Model Analysis Framework (MAF) are utilized. For each of the metamodel elements of the EGPPMM, evaluation rules based on attributions specify a data flow analysis (see [152]) that determines all possible paths to the EGPPInitialNode starting from the EGPPFinalNodes. The set of possible paths is calculated in advance based on the fully-blown Test Model. Subsequently, a comparison between the calculated paths and the set of mapped Test Model elements is performed. This comparison can be conducted with varying degrees of severity depending on the test engineer's preferences (retention policy). I.e. a path is included in the result set if only one model element of the path has been mapped (require_one), or it even requires that all model elements of the considered path have been mapped (require_all). The graph structure described by the resulting set of paths, which is determined according to the selected retention policy, determines the Reconstructed Test Model. Schematically illustrated in figure 8.4.

The final task of this process step is the application of the Test Model specific configuration parameters, namely the specified test levels. As explained in section 8.1.1, the traditional test levels Unit, Integration, and System are utilized. In our schematic representation (see figure 8.4), for example, the Unit and Integration levels are specified as the target test levels, i.e. the Filtered Test Model is composed of one Integration Test Model and the two Unit Test Model components. The model parts that tackle higher integration levels are grayed out.

In addition to the conceptual description, the procedure for this process step is shown utilizing the Running Example. The Filtered Integration Model determined in the last section serves as a starting point (see figure 8.3 left side). However, the details are left out, but an illustration including a brief explanation of the results of the steps as shown in figure 8.5 is conducted.

In the middle of the figure, the Reconstructed Test Model is shown, which is calculated based on the retention policy require_one, a relatively soft criterion for the path mapping and reconstruction phase. After applying the test levels Unit and Integration, the Filtered Test Model is obtained, which is located on the right side of the figure.

8.2.3 Test Model Split and Enrichment

The last remaining process step in the context of Test Model Scoping is the *Test Model Split and Enrichment*. During processing, the Filtered Test Model of the *Test Model Map-*

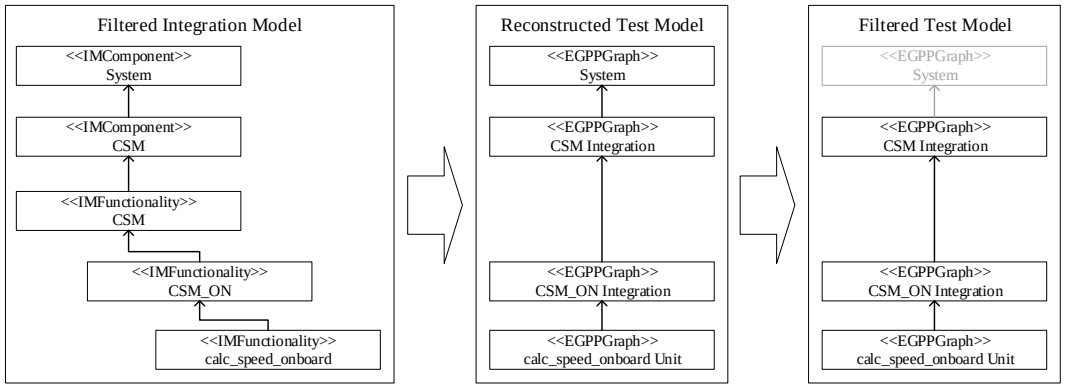


Figure 8.5: Algorithmic approach for *Test Model Mapping and Reconstruction* applied to the Running Example

ping and Reconstruction step is split into independent Test Models of uniform integration level and thus do no longer include any hierarchy information. In this step, the set of resulting Test Models can be enriched with information from the Omni Model. This is especially beneficial if this information can be used profitably by external test case generators.

The concepts are illustrated by a schematic representation (see figure 8.6). On the left side of the figure, the starting point is given by the Filtered Test Model.

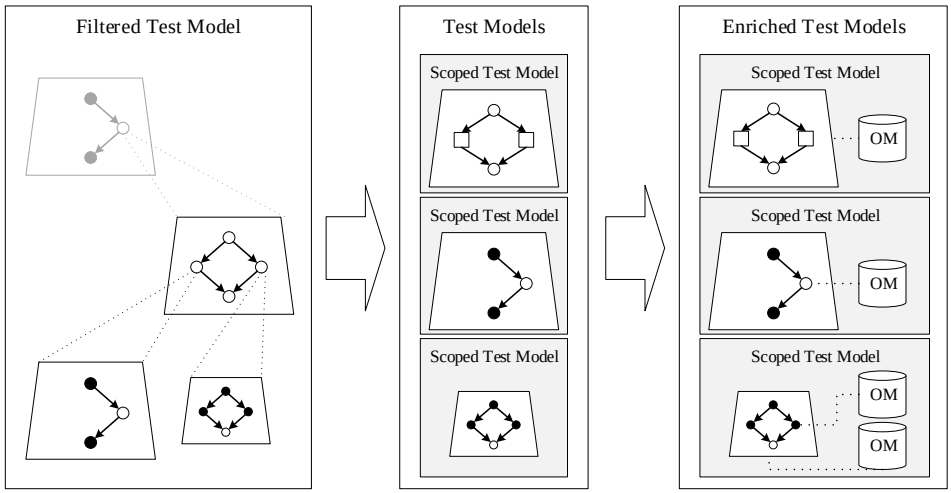


Figure 8.6: Algorithmic approach for Test Model Split and Enrichment

The model artifacts excluded from the fully-blown Test Model are again displayed in gray color. For the generation of test cases and their subsequent execution, however, it is difficult when test cases of different integration levels appear mixed.

Therefore, the goal of the Filtered Test Model decomposition is to create a set of Test Models that each represent a uniform integration level and can therefore be considered in isolation. I.e. the connection between model elements of the sub-model of the higher integration level and the sub-models of the integration level below have to be suitably

resolved. In the context of the EGPP-based representation of the Test Models, such connections are modeled by an `EGPPGraph` node in the control flow of the sub-model of higher integration level and a graph contained therein, which describes the sub-model of the integration level below. This can be achieved by extracting the model elements of the included graph and keeping only the stub model information on the `EGPPGraph`. The concepts of these stub models have been discussed in section 7.2.2, a concrete application is shown in chapter 10. In the middle part of figure 8.6, these stub models are visualized by squares included in the control flow. The model elements extracted from the graph form an independent Test Model. In total, three independent and semantically correct Scoped Test Models are created from the three original sub-models of the Test Model.

After the set of Scoped Test Models is determined, it is possible, to annotate the model components with additional information from the Omni Model. In this way, information available within the MCSTLC can be made available for external tools. This can improve the quality of the resulting test cases significantly and the necessary documentation, e.g. the generated test reports, can be improved. The construct `EGPPTaggedData` is used to implement a flexible annotation mechanism. In particular, it is possible to specify the information for individual model elements of the Test Model. Furthermore, additional information can be annotated for the entire `EGPPGraph` of a Test Model. In figure 8.6 the dotted lines in the right area indicate how the information is assigned. An example of such data is linked requirements that justify the linked tests, or quantification of investigations regarding the safety evaluation of certain system parts, respectively the relevance of the assigned test cases.

Finally, this section refers to the Running Example by applying the steps just presented to the intermediate Test Model of the previous section. Figure 8.7 shows on the left side the initial situation and on the right side some relevant sections of the resulting *Scoped Test Models*.

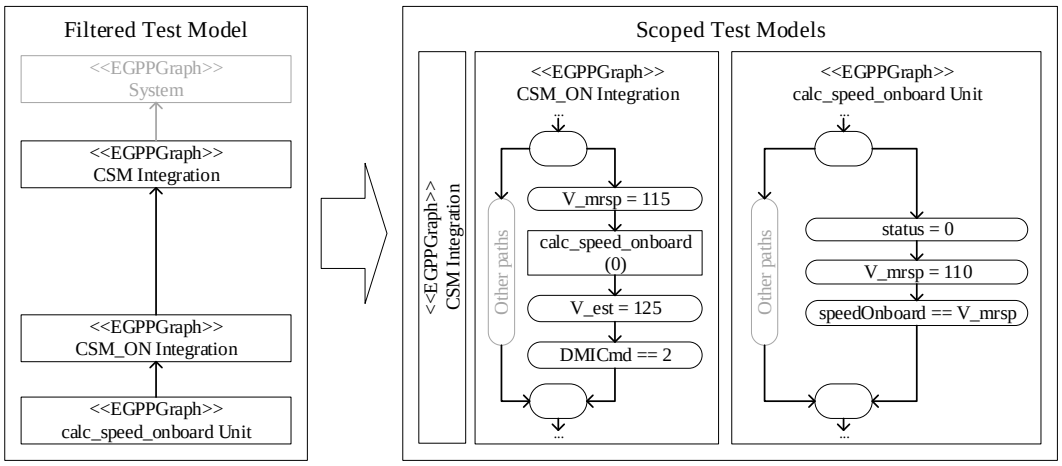


Figure 8.7: Algorithmic approach for *Test Model Split and Enrichment* applied to the Running Example

The Test Model `CSM Integration` is included in the figure for completeness, but con-

tains no information about the included test paths. In contrast, the `CSM_ON Integration`, as well as the `calc_speed_onboard Unit` Test Model give insight into details. The first one shows an example path from the set of paths (test cases), which includes a call of the method/function, tested by the second Test Model. Moreover, the stub model is introduced, visualized as a rectangle. The nodes with rounded corners are `EGPPNodes`, containing atomic statements of the resulting test cases. This way, a large number of tests are mapped in the model. In figure 8.7 these remaining parts of the graph are represented by the nodes with the labels "`|other paths|`" or "`|...|`". The same holds for the second graph `calc_speed_onboard`, which, unlike the other graph, does not contain any stub models.

8.3 Technical Realization within A3F

Based on the Architecture And Analysis Framework (A3F) presented in section 7.3, extensive analyses can be realized on models. This is especially true for the process steps of the MCSTLC, implemented either as stand-alone analyses or as a combination of several analyses. The same holds for the Test Case Management functionality and the algorithmic implementation through TMS.

Using so-called Configurations, both the dependence on other analyses and the necessary parameters can be passed to the A3F. For the analysis `im_scoping` up to five different parameters can be specified, which in turn control the behavior of the internal functionality. Table 8.1 lists the possible parameters and gives a brief explanation in each case.

Table 8.1: Configuration parameters for `im_scoping` analysis

Parameter	**Description**
`integrationmodel`	Determines the input set of Omni Model elements for processing
`customrootguid`	A `GUID` of an Integration Model element, which represents the new root element of the *Integration Model Based Filtering* step
`aspectconstraints`	An Aspect Constraint Language snippet to be evaluated during the *Integration Model Based Filtering* step
`retentionpolicy`	Determines the retention policy applied after the data flow analysis for Test Model paths generation during the *Test Model Mapping and Reconstruction* step. Possible values are `require_one` or `require_all`
`testlevels`	The target set of test levels, which determine the *Filtered Test Model* as a result of the *Test Model Mapping and Reconstruction* step

Besides the necessary model information, which is specified by the parameter `inputs`, there are some implicit dependencies to other analyses (see figure 8.8).

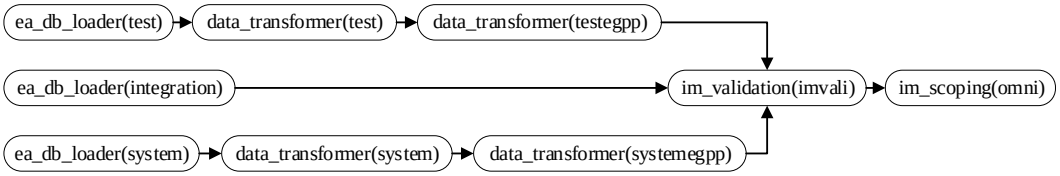


Figure 8.8: Analyses dependency graph for the `im_scoping` analysis

On the one hand, analyses are necessary, which provide the model information, like `ea_db_loader`, on the other hand, these must also be transferred into the required representation format (e.g. EGPP). In particular, it is necessary for the `im_scoping` analysis to perform the `im_validation` analysis in advance. At this point, the model information around the Integration Model is checked and the links of the Integration Model to the other Domain-Specific Models is finalized.

Listing 8.2 shows an exemplary configuration of the A3F, whereas the preceding analyses are not detailed at this point.

```
<configurations of preceding analyses>

analysis im_scoping(omni) {
  integrationmodel=
    "data_transformer|integration|DataTransformationResult|getOutputElements";
  customrootguid="";
  aspectconstraints=
    "partID:in ['CSM'] & safetyLVL:gre ['3'] & devPRIO:gr ['7']";
  retentionpolicy="require_one";
  testlevels="unit,integration";
}
```

Listing 8.2: Example configuration for the `im_scoping` analysis

If this configuration is executed in the context of the Omni Model of the Running Example, the results match with the ones shown and explained in section 8.2.3.

8.4 Related Work

To better classify the concept, several approaches that have addressed the challenges of *Test Suite Reduction*, *Test Case Selection*, and *Test Case Prioritization* are presented in this section. However, it should be noted that the majority of the described approaches are based on the code-level or on an existing set of concrete test cases, which makes the comparison to our approach somewhat abstract.

The range of MBT approaches is very diverse, which is reflected in the publications in this area [169][57]. In particular, the challenges mentioned above usually represent criteria against which different approaches are compared. Utting et al. have differentiated in their taxonomy for example about the applied *Test Selection Criteria* [169]. The

included approaches clearly show that algorithmic attempts to determine an appropriate test suite are the preferred option. However, in the context of their *Conclusion and Outlook*, the future research question of a meaningful "domain-specific test selection criteria" [169] is proclaimed. In our opinion, such a criterion is based on an extensive domain-specific knowledge base and cannot be solved sufficiently by purely algorithmic approaches.

A typical application case is given by Hemmati et al. in the form of an industrial case study [88]. When implementing a Hardware-In-The-Loop setup in conjunction with MBT the phenomenon of test case explosion is frequently observed, which makes an effective mechanism for test case selection indispensable. The consideration of different approaches to select an effective set of test cases led to a genetic algorithm that eliminates similar test cases and thus reduces the resulting test suite. Comparing this approach with our Test Case Management approach, this type of decision-making can be mapped by the Test Model Scoping mechanism. By an upstream analysis of the Test Model, similarity values (fitness function) can be derived and annotated to the model in the form of Aspects. However, in determining this quantification, a much more comprehensive source of information can theoretically be used, which can offer a significant advantage over test case analysis. In addition to the selection of test cases, prioritization can at the same time be achieved.

In the sense of our mapping of sub-models to models of other domains, some approaches in the literature are based on the most obvious artifacts in the development process, namely the requirements. Harrold et al. use the link between requirements and concrete test cases to control the resulting set of test cases. [85] In connection with the concept of tracking requirements across different model artifacts, as presented by Abbors et al., a concept results which is comparable to our generalized approach to linking and selection based on information from a diverse set of domains. [13] An alternative approach for prioritizing test cases based on requirements information is presented by Arafeen et al.. [23] Based on the relationships between requirements, code fragments, and test cases, clusters are formed, which in turn are prioritized based on heuristic analysis. Such a prioritization via clustering can be realized by our approach by including information from System Modeling, Test Modeling as well as requirements modeling. Using a corresponding metric, which is quantified by an aspect, the algorithm can then automatically create the appropriate Scoped Test Models. A similar approach is presented by Abbas et al. which in turn prioritizes requirements and then maps them to the linked test cases. [12]

In the context of selection and prioritization of test cases, approaches are frequent in the literature that in one way or another use costs as a basis for decision-making. Herzig et al. for example select and prioritize based on the execution costs for test cases. [89] To estimate/minimize possible follow-up costs, risk-based testing or the selection and prioritization of test cases based on risk assessments is often applied. [66][65] Such test case management approaches are of particular relevance in the context of regression testing, where the focus is on reducing costs. Therefore, Khatibsyarbini et al. have published a comprehensive literature review, which includes the different decision criteria and respective trend analyses. [108] The same holds for the research of Engstrom

et al., which in particular investigate the two aspects of *Cost reduction* and *Fault detection effectiveness* of the mentioned approaches. [60] Due to the model-centric character of our approach, all information about the evolution of the model basis is available, which allows the implementation of a variety of the mentioned regression test selection approaches through our test case management approach.

Overall, the conducted literature research in the field of selection, prioritization, and reduction of test cases could not identify a matching approach. The conformity of certain sub-concepts in the area of cross-domain evaluation of development artifacts, as well as the feasibility of other solution concepts within the scope of our approach, indicate the meaningfulness and relevance of the developed concepts.

8.5 Conclusions and Outlook

In the course of this chapter, we have presented an approach for Omni Model-Based Test Case Management. This approach tries to incorporate as much information as possible from the comprehensive Integrated Model Basis into the process of Test Model Scoping. Based on that, many targeted Test Models matching the previously specified focus of the test engineer, are generated. The question formulated in section 1.1

How would an intuitive and holistic data-driven test case management approach (selection, prioritization, and reduction) on the model-level look like leading to manageable test suites reflecting the tester's mindset?

can therefore be answered as follows.

By combining a variety of information sources in the context of the Omni Model, the approach presented provides a comprehensive and solid data basis. In combination with the Aspects concept and the language constructs based on it, this basis can easily be extended with meta-information. Furthermore, any information can be used intuitively and consistently by users in this way. The flexibility of these concepts makes it possible to cover several disciplines of Test Case Management, such as Selection, Prioritization, and Reduction with the same approach. In each case, the problem is reduced to the definition, specification, and selection of appropriate Aspects, which in turn are under the control of the user. Possible application scenarios such as *Risk-Aware Test Case Prioritization*, *Model-Based Regression Test Selection*, or *Product line-Aware Test Case Management* have already been demonstrated in [145].

Each approach has advantages as well as shortcomings, discussed below. One of the advantages of this concept is undoubtedly its ease of use. This is reflected by the intuitive interface and adaptability for the user and by the automated processing of the model data. Especially the final processing step, which provides an export to different formats, is a clear advantage since context-specific external processing chains can benefit from the information base. The most important advantage, which is in the sense

of the overall question of this thesis, is the reduction of complexity. Specifically, the task of selecting test cases is shifted from an almost unlimited number of test cases to the model-level, which increases the degree of abstraction and makes interrelationships more tangible.

However, the advantages mentioned above are counterbalanced by some shortcomings, which are important when weighing up practicability. As already mentioned in the context of the Integrated Model Basis, the quality of the domain-specific models and especially of the Integration Model plays a central role throughout all process steps of the MCSTLC. If the models are not created by the formulated guidelines and are not maintained throughout, this has a significant impact on results that are determined in a largely automated manner. Another shortcoming of the presented approach to Test Case Management is the rather sequential nature of the test process. Usually, the mentioned disciplines are performed after the creation of the concrete test cases. This can lead to a reduced acceptance of the approach by the user.

Altogether, the approach described in this section represents an essential part of the MCSTLC. Possible improvements of this process step are linked to the mentioned shortcomings. On the one hand, learning algorithms could be used to complete and improve the Integrated Model Basis concerning the Aspects used or the linked domain-specific models. On the other hand, this would improve the quality of the results, but is related to some pitfalls, whereby a supposed improvement of the information base leads to a deterioration of the results. Further, a survey of model-based developers and testers could provide valuable insights into the best possible integration into existing processes. A corresponding adaptation could significantly improve the acceptance of this approach.

9

Model-Based Abstract Test Generation

The next step of the developed MCSTLC includes the generation of test cases from Test Models. For this purpose test cases are derived from the Scoped Test Models determined in the previous processing step. To provide the algorithmic solution with sufficient information, the Omni Model (Integrated Model Basis) is utilized, as well as expert knowledge to configure the process step. By integrating all the information, it is possible to create very targeted test suites for submodels of the SUT. Besides, the processing of the information is automated. The expert is only necessary for the readjustment of configuration parameters and the monitoring of the processes. Furthermore, the process step of test case generation can be applied in different contexts (see figure 9.1).

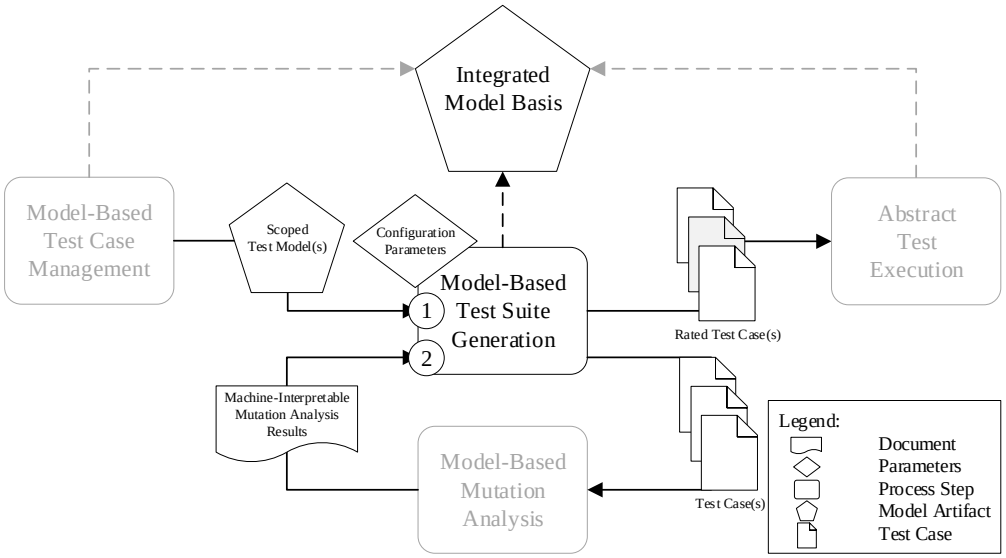


Figure 9.1: MCSTLC extract focusing the Test Case Generation and involved information

The process step can be applied to the first iteration of the test case determination of a specific Scoped Test Model (see the upper part or 1). I.e. no test cases have been created yet for the Test Model, and therefore the quality of these test cases has not yet been evaluated. In this case only the model information and the configuration parameters of the experts can be used.

Moreover, Test Case Generation can incorporate information about an already analyzed test case set of the respective Scoped Test Model (see the lower part or 2), obtained from the Omni Model-based Mutation Analysis (for details see chapter 11). Therefore, an evaluated test case set is available, which can be reduced according to various criteria (e.g. Mutation Score) and ultimately represents the determined test suite. In particular, it can be used to determine a better starting point for the test cases at the code-level and to design the test suite for the subsequent test case execution as efficiently as possible.

9.1 Prerequisites for Test Suite Generation

The number of Scoped Test Models plays an important role in the MCSTLC process, as discussed in detail in chapter 8. In addition to the Scoped Test Models, the expert plays an essential role, since she/he has to specify the starting point of the automated processing chain as well as the parameters for determining the quality of test cases (e.g. a threshold for the Mutation Score) and for defining the exit criteria.

9.1.1 Expert's Configuration Parameters

These configuration parameters are divided into different areas, each of which relates to the configuration of specific parts of the process step.

Start Criterion

In context 1, the definition of a suitable *Start Criterion* for the generation of an initial set of test cases represents one of the expert's core tasks. Starting from this initial criterion, all iterations of the MCSTLC can be optimized concerning the defined exit criteria, which is described in more detail in section 9.2.2. To define such a Start Criterion, the expert has to assign a criterion of the subsumption hierarchy (see figure 5.5) to the respective Scoped Test Model. Besides, she/he can configure the algorithm that implements the selected criterion with a specific amount of information, e.g. a maximum number of derived paths. In many cases, such information improves the performance of the considered Scoped Test Model.

Exit Criteria

Further, appropriate *Exit Criteria* determine whether the generated test suite meets the expert's requirements and thus the continuous process of improving the test suite can

be terminated. The mentioned criteria are specified on the domain-specific model artifacts involved in the Omni Model and evaluated in the context of section 9.2.1. Which domains are included in the evaluation is determined by the expert. Further, the expert determines per domain which coverage metric is to be applied and which threshold the metric needs to exceed. The set of applicable metrics includes the already mentioned coverage metrics but can be extended by special metrics from the respective modeling domain. In the context of the MCSTLC, however, each metric involved must be identifiable.

Test Case Quality Threshold

The expert defines a threshold value for the test case quality, which is based on its Mutation Score (see section 11.2.3). If a test case from the set of generated test cases does not meet this threshold, it is excluded from the final test suite. The choice of the concrete threshold value has to be realized based on empirical values. This is necessary since the application of a dynamic threshold value in early iterations of this feedback cycle would have negative effects on the resulting set of test cases, which is due to missing knowledge of the average quality of the initial test suite. However, an adjustment of the threshold value by the expert during later iterations is conceivable.

9.1.2 Machine-Interpretable Mutation Analysis Results

In contrast to the input artifacts that have to be created manually, this artifact is the result of the Mutation Analysis (see chapter 11). In particular, all information on Mutation Analysis of the previously generated test cases, such as the Mutation Score or the execution reports, is collected here. The mentioned execution reports include details on the respective executions of a test case against the set of created mutants. This enables the expert to reconstruct the way the test case produced the respective test results. In addition to the machine-processed parts of the report, a concise, human-readable version of the information is provided, giving the expert insight into the automated processing, which can be used to optimize the manual configuration parameters.

9.1.3 Excerpt of the Omni Model

In the context of this process step, the Test Model as part of the Omni Model is used to derive test cases. Further, connected domain-specific models are utilized for the evaluation of metrics, which are part of the exit criteria of test case generation. Since the exit criteria may just be defined on the System Model, an Omni Model consisting of Test Model, System Model and Integration Model represents the minimum configuration for this process step. However, it is recommended to incorporate more domain-specific models to be able to detect model smells at an early stage of development.

9.2 Test Suite Generation

Based on the input artifacts introduced above, the current process of test case generation is explained in this section. The process is divided into four sub-steps.

The first step of the process called *Artifact and Feedback Evaluation* analyzes the artifacts necessary for test case generation. In particular, this step has to be seen in the two contexts introduced in figure 9.1. If Scoped Test Models are under consideration, which have not been processed in any of the previous iterations, these models are evaluated for the chosen Start Criterion. Otherwise, the evolution of several indicators for the previously derived test suite are tracked. This includes the evaluation of the multi-domain coverage as well as the identification of bad test cases in terms of their Mutation Score, usable as a criterion for reducing the test suite.

The second step of the process, *Test Case Generation Metric Adaption*, deals with the selection and adaptation of the criterion and the associated algorithm, to obtain a set of test cases from the Scoped Test Models. If no information from previous iterations is available for the Scoped Test Models, expert knowledge can be used to determine an initial criterion. In case of no such information being available, the Start Criterion is determined from the information collected in the *Artifact and Feedback Evaluation* phase. Based on the evolution of the multi-domain exit criteria metrics over the last iterations, a new criterion is selected, which in turn determines the algorithm applied to generate the test cases.

The third step of the processing chain called *Data Flow Analysis-Based Test Case Generation* deals with the conceptual implementation of the test case generation using data flow analysis techniques. In particular, the relationship between the criteria of the subsumption hierarchy and the associated algorithms is detailed in the form of MAF-based analyses for EGPP-based Scoped Test Models. An analysis in Model Analysis Framework (MAF) comprises a data flow analysis of an EGPP instance, which evaluates any data annotated to the metamodel.

The fourth and final step of the process, *Feedback-Oriented Test Suite Creation*, incorporates the lessons learned from previous iterations on the relevant Scoped Test Models. In particular, quality properties of individual test cases are projected onto test cases of the newly created test case set, thus implementing a kind of blacklisting for bad test cases. This reduces the final test suite to an efficient subset of the generated test cases.

9.2.1 Artifact and Feedback Evaluation

This processing step is the starting point of the test case generation. On the one hand, the input models and on the other hand the feedback of an already performed mutation analysis of these models are processed here. The former is particularly useful if

no start criterion is specified and no knowledge of derived test cases is available yet. In particular, the Scoped Test Models are evaluated to gain first insights into suitable coverage criteria for the subsequent selection process. This evaluation looks at the nature of the graph structure that defines the respective Scoped Test Model. Depending on the complexity of the initialization phases, the degree of branching, and the occurrence of loops included in the Scoped Test Model, a Start Criterion can be chosen apart from *Node Coverage*. However, the selection is limited to criteria that represent leaves of the subsumption hierarchy shown in figure 5.5, which maximizes the potential for improvement This is mainly because the iterative improvement of the resulting test suite across several runs of the MCSTLC provides a higher number of variation possibilities.

However, the focus of the *Artifact and Feedback Evaluation* step is on advanced iterations of the MCSTLC (see 2 in figure 9.1). In this context, the findings of the Mutation Analysis are used to reduce the number of test cases by excluding test cases with a poor quality rating. Currently, the decision is based on the Mutation Score of the test case but may include other aspects in future versions. Together with the *Test Case Quality Threshold* specified by the expert, which represents the minimum Mutation Score to be achieved, the selection is determined. The test cases with a Test Case Mutation Adequacy Score (TCMAS) greater than the threshold are included in the test suite, while the others are put on a so-called blacklist and do not contribute to the test suite as long as the threshold does not change. Excluded test cases from previous runs remain on this blacklist if no parameter adjustments are made. This step aims to be able to make a statement about the quality of the resulting set of evaluated test cases. In particular, the subsequent execution of the test suite is therefore improved by reducing the total number of tests to be executed by test cases with low significance.

In addition to the quality of the individual test cases, the entirety of the test cases in this process step is examined concerning the specified *Exit Criteria*. These are specified by the expert and often reflect external requirements from the development process. This can be a regulatory requirement of a standard that specifies a special form of coverage for appropriately evaluated system parts. In the automotive industry, for example, the standards *ISO 26262* or *IEC 61508* require statement coverage, branch coverage or MC/DC coverage depending on the Automotive Safety Integrity Level (ASIL) classification of the system part [5]. As figure 9.2 shows, a specific metric can be defined for each modeling domain, including a threshold value for compliance.

Due to the relationships between structural as well as behavioral components defined in the Integration Model, the evaluation of the metrics of the respective modeling domains can be performed, driven by the determined set of test cases. Regarding the procedure for evaluating the metrics, a distinction is made between the System Modeling domain and the other domains. The former is based on the system paths triggered by the test cases and their ratio to the total of possible system paths. The second one evaluates the connections between model elements of the considered test cases and model elements of the respective modeling domain. An element that specifies a connection to an element of a test case is considered to be covered and is thus included in the evaluation of the specified metric. As soon as all metrics specified in the *Exit Criteria* are met

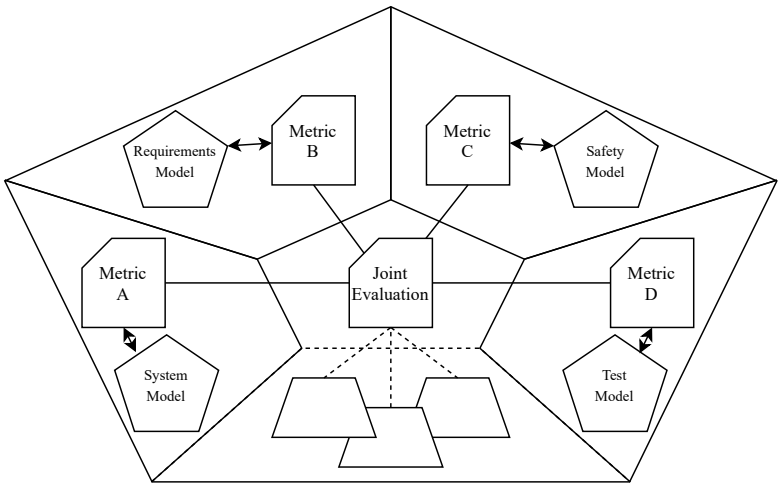


Figure 9.2: Exit Criteria specification via multi-domain metric evaluation

by the set of test cases concerning the thresholds, the iterative process is aborted. Otherwise, the process is continued and, if necessary, adjustments are either made to the Coverage Criterion for test case generation or the *Exit Criteria*. The following section explains how the evaluation of the *Exit Criteria* impacts the adjustment of the *Coverage Criterion*.

In the course of the explanation of this processing step, the Running Example is used for illustration. Here, the parameters specified by the expert are selected as follows.

```
startcriterion = "|nodecoverage|"
tcmasthreshold = "0.2"
exitcriteria = "system|nodecoverage|0.8"
```

According to the specified parameters, the *Node Coverage* criterion is suggested as a starting point for all Scoped Test Models that have been determined. In the iterations for selecting high-quality test cases, only test cases with a TCMAS greater than `0.2` should be included in the resulting test suite. The *Exit Criteria* was specified in this case as the sole criterion for the relevant System Model, according to which an `80%` *Node Coverage* must be achieved. Provided that for the Scoped Test Models of the Running Example the *Exit Criteria* is completely fulfilled after a certain number of iterations, the final test suites have been determined.

9.2.2 Test Case Generation Metric Adaption

Based on the information obtained from the analysis of the Scoped Test Models or the evaluation of the feedback from the Mutation Analysis, a suitable coverage metric can be determined for the subsequent test case generation. To understand the procedure

explained in the further course, a short preview of the realization of test case generation through data flow analysis is given here. Whenever we talk about a coverage criterion that is used for test case generation, we implicitly refer to a corresponding data flow analysis of the MAF that produces a test suite that covers the associated criterion best. In particular, this can be done for all the criteria shown in figure 5.5. If each of the criteria specifies a corresponding data flow analysis, the decision space for the criteria adaptation process is obtained, which is shown schematically in figure 9.3.

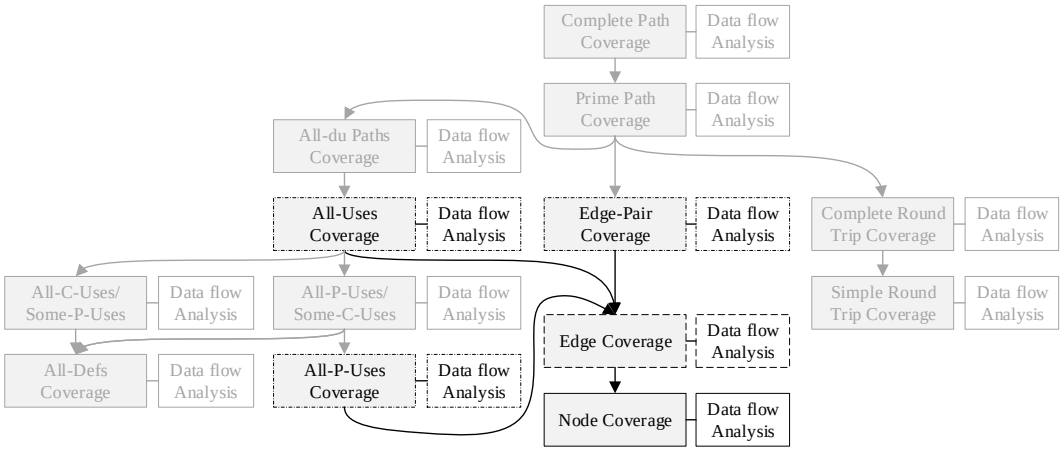


Figure 9.3: Coverage Criteria Subsumption Hierarchy including connected data flow analyses (based on figure 5.5)

The entry points for test case generation determined in the previous section usually represent the leaves of the illustrated tree structure. An exception can be the start criterion manually specified by the expert since it is possible to select any criterion of the tree structure. Such a *Start Criterion* may quickly lead to a test suite that meets the specified *Exit Criteria*, but might be not optimal concerning the amount and complexity of included test cases.

Therefore, when selecting the Start Criterion by analyzing the relevant Scoped Test Model, the *Node Coverage* criterion is usually set as default. This criterion is one of the weakest criteria since a high degree of coverage can usually be achieved by a relatively small number of paths through the EGPP Test Model. However, it is not possible to draw any conclusions about the respective coverage of the connected EGPP System Model. This in turn is one of the advantages of having these models created by different experts, since the first differences between the target and actual system are revealed early. Once a Start Criterion is determined, the data flow analysis is applied, which is discussed in the following section.

In the context of further iterations of the MCSTLC (see 2 of figure 9.1) the feedback is furthermore included. In this case, the hierarchy introduced above comes into effect. However, before the algorithm for the evolution of the applied coverage metric is discussed, another quantity is introduced, which is used for decision-making. This measure considers the evolution of the *Exit Criteria* over the last two iterations of the MCSTLC. In particular, a so-called `metricsTrend` is determined, which indicates whether

Iteration \ Domain	system	requirements	safety
0	0.2 (edge cov.)	- (node cov.)	0.3 (node cov.)
1	0.35 (edge cov.)	0.9 (node cov.)	0.32 (node cov.)
δ	+0.15	+0.9	+0.02

Table 9.1: Example *Exit Criteria* evaluation to determine the metricsTrend

the last adjustment of the Coverage Criterion for test case generation has lead to an improvement. For this purpose, the change of each atomic criterion involved in the *Exit Criteria* is examined and the median of the changes is further processed as metricsTrend. To show this by a concrete example, the following evaluations of the *Exit Criteria* of two past iterations of the MCSTLC are utilized.

As the table shows, the values of the *Exit Criteria* have improved throughout. For the modeling domains system and safety by +0.15 and +0.02 respectively. For the modeling domain requirements even by +0.9, which is since after the iteration 0 the *Exit Criteria* was adjusted by including the respective domain. This is the reason why instead of the mean value (+0.357) the median (+0.15) is chosen to determine the metricsTrend, because it better represents the actual change in such a situation.

To select a better criterion for the following step, three ranges are defined on the scale of the metricsTrend variable (from -1 to 1), which are used to determine the future criterion using different methods (see figure 9.4). The values (-0.02 and +0.02), which determine the three ranges, are only to create a certain blur around the neutral value 0 and do not have any special meaning.

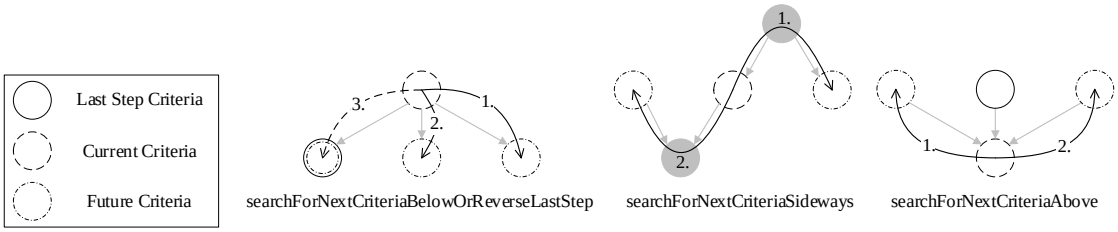


Figure 9.4: Ways of determining the future Coverage Criterion applied

On the one hand, the range from -1 to -0.02 is used as an indicator for poor performance in the previous step. In this case, the future criterion is selected by the method searchForNextCriteriaBelowOrReverseLastStep. The method searches for a new criterion at a lower level, giving preference to criteria that were not used in previous steps. However, if there is no alternative, the criterion of the previous step can be reselected, which stops the adaptation process.

The second range covers the values between -0.02 and +0.02, which reflects a weak increase/decrease in the value of the metric. Such values are seen as an indication of stagnation of the increase/decrease, so in this case, the method searchForNextCrite-

`riaSideways` provides the future criterion. As you can see in the graph, it is possible to determine criteria on the same level by utilizing lower as well as higher-level elements of the structure.

The third and last area ranges from `+0.02` to `+1` and reflects very positive development in the past choice of the coverage criterion. In such a situation, we try to carry on the positive trend and determine the future metrics by the `searchForNextCriteriaAbove` method. This method moves up in the structure but excludes criteria that have already been applied in previous steps. If no new criteria are found, the process ends, because either the top-level of the structure has been reached, or all higher criteria have given worse values for the *Exit Criteria*.

Regardless of the method that produced the set of future criteria, a final criterion is randomly selected. For our Running Example, we assume that the *Node Coverage* criterion has initially (`Iteration 0`) been selected, and in the current step (`Iteration 1`) the *Edge Coverage* criterion is evaluated. Based on the `metricsTrend` determined in table 9.1 `+0.15`, the algorithm uses the method `searchForNextCriteriaAbove` to determine the set of future criteria. If we search for new criteria in figure 9.3 according to this method starting with the *Edge Coverage* criterion, we obtain the criteria *All-P-Uses Coverage*, *All-Uses Coverage*, and *Edge-Pair Coverage*. The criterion selected from this set is used in the next step of the processing chain to perform the connected data flow analysis. Details about this processing step are conducted in the following section.

9.2.3 Data Flow Analysis-Based Test Case Generation

Based on the criterion selected in the previous step, this section deals in more detail with the determination of a concrete test case set using data flow analysis techniques. As already shown in figure 9.3, a data flow analysis is specified and linked for each coverage criterion. Specifically, this is performed at the model-level, utilizing concepts of the MAF [152]. For this purpose, the respective metamodel is provided with data flow attributes. These attributes are assigned concrete values, which are evaluated afterward. The analysis-specific EGPPMM of the MCSTLC, which is used for the internal representation of the System Models as well as for the Test Models and Scoped Test Models, has already been discussed in section 7.2. For this reason, all data flow analyses that are implemented in the context of MCSTLC are based on this metamodel.

In conjunction with the data flow attributes, which are annotated to the metamodel, update rules are specified for the contained model elements. The evaluation of the update rules considering the attributes starting from the respective `EGPPInitialNode` results in a set of paths through the specified graph structure. The subsequent selection of paths ending in an `EGPPFinalNode` implicitly reflects the resulting test suite. In this way, unreachable parts of the specified graph structure are excluded from the final set of paths, and only valid runs through the Test Model are considered.

To better illustrate the conceptual explanations, the application context of the Running

Example from the previous section is taken up again. In particular, the `Edge Coverage` criterion mentioned above are discussed about the set of update rules and the data flow attributes. In contrast to the Object Constraint Language (OCL)-based specification of update rules, as shown in listing 7.3, this section utilizes a pseudo code-based representation, because the original and comprehensive specification would be too extensive.

Algorithm 1 MAF-based data flow analysis for the edge coverage criterion

```
function updateRuleEGPPInitialNode(node)
    return new HashSet<List<EObject»()

function updateRuleEGPPTransition(transition)
    result = new HashSet<List<EObject»()
    sourceNode = transition.getStartNode()
    previousPaths = getPathsFromNode(sourceNode)
    if sourceNode instanceof EGPPInitialNode then
        result.addPath(sourceNode)
    else
        for incomingPath ∈ previousPaths do
            if maxTraversalTransitionsReached(incomingPath, transition) then
                continue
            if transition.getEndNode() instanceof EGPPFinalNode then
                updateAndCheckEdgeCoverageStatus()
            result.addPath(incomingPath.extendPath(sourceNode)
    return result

function updateRuleEGPPNode(node)
    result = new HashSet<List<EObject»()
    for incomingTransition ∈ node.getIncomingTransitions() do
        previousPaths = getPathsFromTransition(incomingTransition)
        for incomingPath in previousPaths do
            result.addPath(incomingPath.extendPath(incomingTransition)
    return result
```

As mentioned above, update rules are specified for all metamodel elements that describe the control flow. In the illustrated data flow analysis for the *Edge Coverage* Criterion, the update rules are limited to updateRuleEGPPInitialNode, updateRuleEGPPTransition, and updateRuleEGPPNode. The update rule updateRuleEGPPInitialNode represents the starting point of each path by initializing a data structure that collects the paths. The other two update rules iteratively build the paths and check the annotated attributes, such as the number of traversals of a transition. Especially the update rule updateRuleEGPPTransition checks if the considered transition is already contained in a path (`maxTraversalTransitionsReached`) and if the set of all transitions is covered (`updateAndCheckEdgeCoverageStatus`). If one of these termination criteria is met, the update rule does not generate a new partial result, or the paths according to the *Edge Coverage* Criterion have been determined completely.

Based on this intermediate result, the paths ending in an `EGPPFinalNode` are extracted, ensuring that only valid runs through the Test Model are included in the resulting set. This way test cases can be derived for the Scoped Test Models shown in figure 8.7. The final set of test cases for the running example is shown in the following section.

9.2.4 Feedback-Oriented Test Suite Creation

Based on the set of paths generated by the respective Test Model, the last step of the test case generation process involves processing the existing feedback. As already shown in section 9.2.1, the findings of Mutation Analysis for test cases from previous iterations are included. In particular, the quality of a test case is estimated and further evaluated to determine test cases for the persistent blacklist.

The focus of this last processing step is to project the blacklist onto the newly created set of test cases. In particular, the final test suite should not contain any test cases of bad quality. However, in the first iteration of the test case generation (see 1 of figure 9.1), the mentioned activities are not applicable and thus omitted. In such a case the test cases previously derived by data flow analysis represent the final test suite. However, in an advanced iteration of MCSTLC (see 2 of figure 9.1), the respective test cases are excluded and the final test suite is determined.

This is shown alongside the Running Example to give a better understanding. Figure 9.5 shows the result of the previously explained processing step.

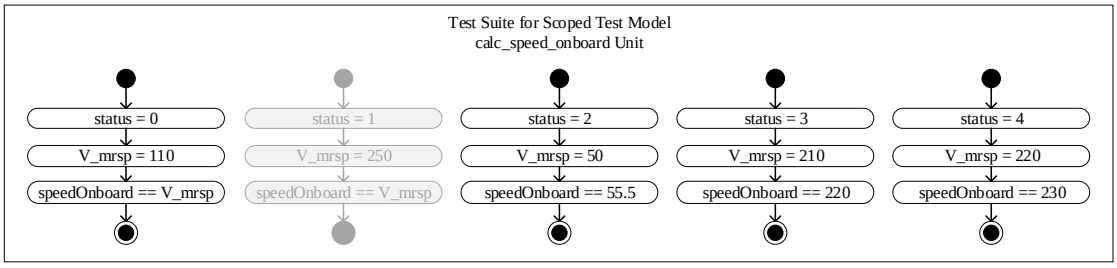


Figure 9.5: Set of generated test cases for a unit-level Test Model of the Running Example

The paths with gray nodes are excluded from the test suite due to an insufficient TCMAS determined in the Mutation Analysis. The paths with white nodes represent the test cases that are either re-subjected to a Mutation Analysis (see chapter 11) or simply executed against the SUT (see chapter 10).

9.3 Technical Realization within A3F

The functionality of this processing step has a prototypical realization within the A3F. For this purpose, the functionality is implemented by an independent analysis of the framework. This analysis encapsulates the previously mentioned concepts and algorithmic sub-steps and offers the user several configuration options.

The configuration options are realized by a series of parameters. Table 9.2 gives an overview of the available parameters, which are specified by the expert on the one hand and given by results of other analyses on the other hand.

Table 9.2: Configuration parameters for `egpp_path_generation` analysis

Parameter	Description
`integrationmodel`	Determines the Integration Model, which contains the mapping information of the corresponding Omni Model
`testmodels`	Determines the set of Test Models for the subsequent test case generation
`blacklistthreshold`	A floating-point value between `0` and `1`, which represents a threshold for the blacklist mechanism based on mutation score
`exitcriteria`	A set of coverage metric thresholds, evaluated on the Integrated Model Basis (see section 9.2.1)
`startcriterion`	Parameter for the specification of a generation metric including metric parameters
`autoadvice`	Determines the feedback data from previous mutation analysis runs

The parameters `startcriterion`, `exitcriteria`, and `blacklistthreshold` require the mentioned expert knowledge and were already explained on a conceptual level in section 9.1 and evaluated in the sub-steps of the process. The parameter `startcriterion` consists of three different parts, not all of them are mandatory. The first part covers the `ID` of the relevant Scoped Test Model, while the second part determines the initial coverage metric and algorithm to be applied for test case generation. The third part allows fine-tuning of the algorithm by modifying internal settings. The parameter `exitcriteria` again represents a dynamic set of triples, where each of the triples defines the evaluation metric for a certain modeling domain included in the Omni Model. Accordingly, the first part of each triple describes the `ID` of the respective domain, the second part the `ID` of the metric to be evaluated, and the third part the threshold, which describes a percentage in the form of a floating-point number between `0` and `1`. The triples are internally linked by a logical `AND`, so the multi-domain metric is only fulfilled when all metrics involved meet their respective threshold. The parameter `blacklistthreshold` is used to define a threshold for the automated selection mechanism, which has already been described conceptually in section 9.2.4.

In addition to the parameters already mentioned, some model information of the Omni Model is required, which is generated in particular by upstream analyses. Figure 9.6 gives an overview of the combination of analyses of the A3F, which are linked to an analysis chain.

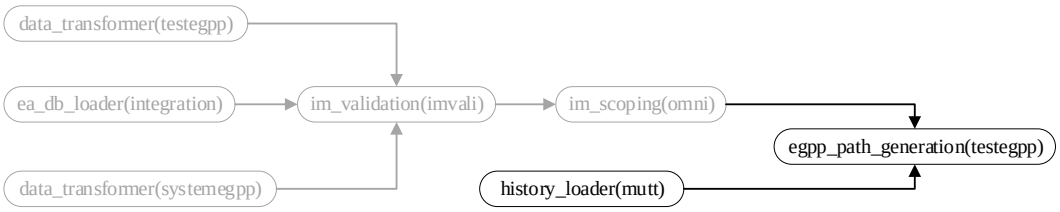


Figure 9.6: Analyses dependency graph for the `egpp_path_generation` analysis

The parameters `integrationmodel` and `testmodels` are taken from the analysis results of the analyses, which are detailed in figure 8.8. This provides the `egpp_path_generation` analysis on the one hand with an Integration Model that contains all connections between the modeling domains involved. All Scoped Test Models, determined in the context of the `im_scoping` analysis are available.

Only the parameter `autoadvice`, which evaluates the Mutation Analysis results of the previous A3F analysis, is a special feature. Technically, the `history_loader` analysis loads analysis results from a previous A3F analysis into the current A3F analysis and can thus be processed further.

To make the descriptions of the individual parameters more tangible, an example configuration is shown for this analysis. (Listing 9.1, preceding analyses are excluded)

```
1  <configurations of preceding analyses>
2
3  analysis egpp_path_generation(testegpp) {
4    integrationmodel="im_scoping|omni|IMScopingResult|getFilteredIMModel";
5    testmodels="im_scoping|omni|IMScopingResult|getGeneratedMBTModels";
6    blacklistthreshold="0.5";
7    exitcriteria="system|nodecoverage|0.8";
8    startcriterion="|pathcoverage|MAX_PATHS:100 MAX_TRAVERSAL_NODES:1
9      MAX_TRAVERSAL_TRANSITIONS:1";
10   autoadvice="history_loader|mutt|EGPPMutationTestingResult";
11 }
```

Listing 9.1: Example configuration for the `egpp_path_generation` analysis

After the analysis has been performed, the analysis result is generated. In this case, lists of test cases are created per Scoped Test Model, where a test case is described by an ordered list of model element IDs, which in turn provides a valid sequence of test instructions. For the continuous Running Example, a possible result of this analysis has been shown in a condensed form in section 9.2.4.

9.4 Related Work

Following the conceptual presentation of our test case generation approach, this section looks at other approaches that are placed in the same context. There are many different approaches in the field of test case generation to overcome the problem of generating a sufficient test suite. Anand et al. show in their comprehensive survey a large part of the available spectrum [20]. This ranges from randomized, search-based, or combinatorial approaches based on development artifacts of the SUD to the model-based approaches that are related to our concepts. To make it easier to classify these approaches, different aspects are focused.

Model Basis

It should be emphasized that the majority of research contributions are based on developmental artifacts of the SUD instead of using their models for testing. In particular, this reflects the large number of UML-based approaches that use and partially extend various structural and behavioral models [157] [128]. For example, Swain et al. have constructed combined models from various artifacts of a UML-based development (Use Case and Sequence Charts), which are used in their processing chain to generate test cases [162]. In addition to the UML-based approaches, variants of timed automata are frequently used, from which concrete test cases are derived through model checking approaches (here UPAAL) [90] [61]. Furthermore, some approaches automatically derive model artifacts from a suitable requirements specification for the subsequent test case generation, which in turn is based on a variant of finite statemachines [163]. In summary, the research contributions are repeatedly based on comparable models, represented as far as possible by the presented EGPPMM, or derived from it. This results in the flexible applicability of our approach to a large number of the model variants shown, provided that a suitable transformation rule between the metamodels is available.

Algorithmic Approach

Based on the respective model artifacts, different algorithmic approaches to derive a set of test cases can be found in the literature. These can be divided into two categories, namely Black-box and White-box. In the first category, there are more and more randomized/combinatorial approaches that cover the input parameters of the SUT to derive a meaningful test suite [119]. In the second category, priority is given to approaches that make profitable use of the knowledge about the internal processes of the system. Anand et al. have investigated some applications of search-based algorithms in this context [20]. In particular, genetic algorithms are frequently utilized to determine the best possible test suite. These algorithms are computationally intensive and therefore offer optimization potential [32]. The form of data flow analysis by Saad et al. used in our approach again describes an unconventional variant of test case generation and

is therefore difficult to compare with the presented alternatives. However, to determine a more effective test suite, Briand et al. applied a data flow analysis that incorporates more context information in addition to the previous control flow view, thus leading to more effective test cases [39].

Test Adequacy Criteria

The last aspect of the presented approaches is the range of criteria that determine whether a generated set of test cases is adequate. As anchored in several standards, many approaches use the classic coverage criteria. For example, Rayadurgam et al. use a set of structural coverage criteria to generate a test suite using model checking [150]. The criteria used in test case generation can be evaluated in combination to increase the effectiveness of the resulting test suite [73]. The criticism of using such adequacy criteria exclusively is clearly expressed by Inozemtseva et al. in their paper [98]. Driven by such critical questioning of coverage-driven test suites, the effectiveness of such test suites is additionally determined by mutation testing approaches [74]. Such a combination of criteria is included in our approach, which makes the quality of the resulting test suite more transparent and resilient.

9.5 Conclusions and Outlook

In the course of this chapter, a promising approach for the adaptive and efficient generation of test cases was presented. Adaptive, in the sense of a continuous adaptation of the algorithms for deriving test cases based on previous results. Efficient, in the sense of a minimum resulting set of test cases, by continuously selecting meaningful test cases. In particular, established methods for the derivation of test cases, knowledge of other process steps of the MCSTLC, as well as the extensive information of the Integrated Model Basis is used. An answer to the question identified in section 1.1

How could better adaptability and context sensitivity of test generation based on criteria and measures on the model-level improve the overall complexity?

looks as follows.

Starting from classical approaches to test case generation as shown in related work, these approaches were first mapped on a special model representation. A more abstract view of the problem can be adopted and on the other hand, a representation independent of the original model can be built upon. Both aspects help to reduce the emerging complexity. Furthermore, seamless integration into the entire MCSTLC is forced, whereby the individual process steps to profit from each other as much as possible. In the context of test case generation at model-level, a continuous adaptation of

the underlying algorithm was implemented, which is primarily based on comprehensible metrics to make the complexity manageable. The findings of the Mutation Analysis are evaluated in terms of a quality measure for test cases and results from past iterations and the Integrated Model Basis are included. All in all, a concept can be implemented which lowers complexity by design and produces results that keep the complexity of the overall process manageable. The focus is on improving classical approaches that just consider the integrity of a test suite to include a measure for the quality of the resulting test suite.

Apart from the positive aspects of the mentioned concepts, a critical discussion of some other aspects is contributed in the following. First of all, the definition of the Exit Criteria based on artifacts of the Omni Model should be mentioned. The possible diversity in the included metrics promises an increased significance regarding the created test suite concerning all modeling domains involved. This functionality heavily depends on the modeling and maintenance of the respective Integration Model elements. If these are not or insufficiently specified, the downstream automated processing chain can produce results that allow false conclusions. Another point concerns the orientation of the approach towards a purely generation-based determination of test cases. This aspect can be weakened such that manually generated test cases can be considered, too. For this purpose, the information must simply be embedded in the relevant Test Models in a suitable way such that they are included in the generation process. For example, in the context of EGPP, this could be isolated paths from an `EGPPInitialNode` to an `EGPPFinalNode`. Finally, the blacklisting mechanism is discussed. At this point, the lack of significance of the Mutation Score, which is determined by the Mutation Analysis, can be pointed out. This score depends on the way the mutants are created for the System Model under consideration. To weaken this shortcoming, the Mutation Analysis detailed in chapter 11 explicitly focuses on a targeted creation of mutants. The presented concept offers the possibility to combine the Mutation Score with further metrics on the quality of the test cases without the need to adjust the remaining processing chain, but improving the basis for decision-making.

From the context of Mutation Analysis, which has so far only been used as feedback in Test Case Generation, it is possible to exploit the potential for improvement in future work on this topic. Besides its application to create mutants of the System Model, the mutation aspect is often used to identify new test cases. This could be an additional component of the strictly rule-based derivation of test cases from the model, in that a kind of fuzzing expands the horizon about new test cases. At this point, the ideas of Exploratory Testing presented in the basic part may be incorporated.

10

Model-Based Abstract Test Execution

In addition to the already presented process steps of the MCSTLC, which cover *Model Creation/Modification*, *Test Case Management*, and *Test Suite Generation*, this chapter presents our concept for the execution of test cases on the model-level. Our approach is intended to improve the applicability of a shift left of test activities towards the model-level. That means the execution of test cases does not require the existence of an executable system in the form of code artifacts but is accomplished using the model artifacts of the Integrated Model Basis. Therefore, the required information must be available in the models, and the models can be transformed into our internal representation (EGPP). To be able to evaluate the former, an evaluation scheme of model artifacts is presented in section 10.1, which evaluates properties of the model and based on that allows a correct classification of the given model, which accordingly influences the processing logic of Abstract Test Execution (ATE). For the latter, a transformation from the present metamodel of the chosen modeling language to our Execution Graph++ (EGPP) metamodel has to be specified. Besides these two prerequisites to be seen in the context of the Integrated Model Basis, figure 10.1 shows the incorporation of the process step into the MCSTLC.

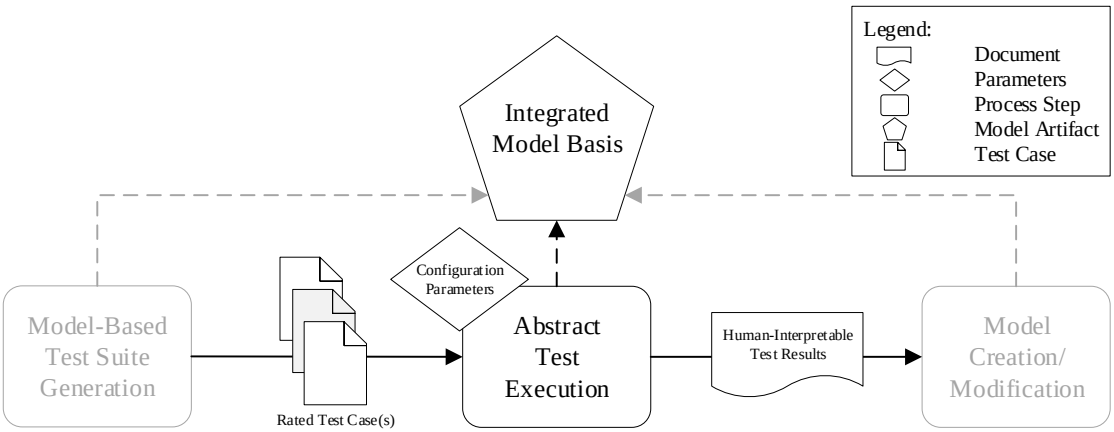


Figure 10.1: MCSTLC extract focusing the Abstract Test Execution and involved information

Another factor is the set of test cases to be executed, as shown on the left side of the figure. This can either be a set of test cases generated from the underlying Test Model, or a derived set of test cases that has already been subjected to a quality assessment

using the Mutation Analysis presented in chapter 11. For processing by the ATE component, this information is transparent but determines the significance of the results. These test reports, in turn, are primarily generated for the expert in the form of *Human-Interpretable Test Results*, whereby improvements to the model artifacts can be subsequently initiated as part of a new iteration of *Model Creation/Modification*. In addition to the input and output artifacts of this process step, the experts are allowed to influence the internal processing logic through a series of configuration parameters.

In addition to the shown incorporation into the MCSTLC, the ATE applies in the context of Mutation Analysis. Here, the execution mechanism presented in the following serves to execute the test cases against the mutated variants of the System Model, which in turn again saves the step towards an executable version of the system on the code-level.

10.1 Prerequisites for Abstract Test Execution

Before going into details about the execution mechanics of ATE, some basic concepts are introduced for a better understanding. The categorization of EGPP-based System Models is discussed, which allows the implementation of the best possible execution technique according to the available information. Moreover, the configuration options and the necessary model artifacts of the Integrated Model Basis are discussed, necessary for a proper execution.

10.1.1 Execution Graph++ Characteristics Analysis

As already mentioned, before performing an ATE, the available System Model is analyzed for its nature. For this purpose, the System Model has to be available in its transformed EGPP variant, which is assumed as given at this point. If the transformations have been specified in advance by the expert, this condition does not pose a problem because it can be performed automatically.

The analysis aims to identify the development stage of the given System Model and the types of information which can be evaluated. In the context of model-based software development, a rough architecture is usually created in a first iteration which mainly specifies structural model elements. Further modeling iterations are used to create coarse behavioral models, which are refined continuously and enriched with detailed information. Our so-called *Graph Levels*, which allow categorization of an existing EGPP, are based on these described phases of model-based development:

- `Level 0:` Fragmented Control Flow Graph
- `Level 1:` Control Flow Complete Graph
- `Level 2:` Data Flow Graph

The first one is the so-called `Level 0`. Under this category, EGPP instances are subsumed that are advanced concerning the structure-giving model elements, but still have fragmented control flow information in the context of the behavior-describing model components. An instance of `Level 0` is therefore further referred to as a *Fragmented Control Flow Graph*. In general, such a categorization is made if the analyzed EGPP instance violates one of the structural or control flow rules of the graph specified in section 7.2.1. For example, a graph that contains model elements that are not passed when traversing any paths from an `EGPPInitialNode` to an `EGPPFinalNode` represents an invalid EGPP instance and is therefore categorized as `Level 0`.

The next level is described by `Level 1`. This includes EGPP Instances that do not contradict any rules of the underlying metamodel to their structural nature and the behavioral descriptions they contain in the form of graphs. Instances of this category are called *Control Flow Complete Graphs*. However, concerning the included detailed information in the form of code fragments of our minimal language presented in code fragment 7.1, these graphs do not exhibit any *meaningful* information. *Meaningful* means that no connections in the sense of *Def-Use chains* can be determined in the given graph.

The last level of this categorization, `Level 2` describes so-called *Data Flow Graphs*. In contrast to the previous levels, EGPP instances of this category fulfill both the requirements of a `Level 1` graph and additionally the presence of a significant amount of detailed information in the included model elements. As far as the detailed information represents valid code fragments of our language in the respective model elements and these correspond to the modeled control flow, the model has reached a level that may serve as a basis for an automated transition towards code. However, code generation is not the focus of our concept but points out the flexibility of our modeling concept.

In addition to using the categorization just introduced in the context of ATE, the Mutation Analysis presented in section 11.2.1 makes use of this mechanism. At this point, the mechanism is used to decide which mutation operators are reasonable and applicable based on the evaluation of the EGPP System Model. This results, for example, in the default set of mutation operators applied by the mutant generation algorithm.

10.1.2 Abstract Test Execution Engine Configuration Parameters

Analogous to the process steps of the MCSTLC already presented, the configuration parameters rely on the experience of the expert as well. That is, the expert can specify a set of parameters that influence the execution mechanism of the ATE concept. Provided that no empirical values are available in the context, the expert can rely on a generic set of default parameters. In the previous section, it was mentioned that the level of the EGPP instance has a crucial influence on the applied execution mechanics in the course of the ATE. Depending on the execution mechanics, different parameters can be configured. Currently, two different execution techniques exist, the *Control Flow-Aware ATE* and the *Data Flow-Aware ATE*, detailed throughout section 10.2.

The *Control Flow-Aware ATE* relies primarily on the structural information of the graph at hand and therefore provides configuration parameters that affect this type of analysis. Specifically, the expert can influence thresholds of the internally used search algorithms such as breadth-first and depth-first search or define termination criteria that make the execution more efficient. For the *Data Flow-Aware ATE*, this results in an analogous selection of parameters that are primarily due to the underlying data flow analysis. Besides, timeouts can be specified to prevent the entire ATE from failing in case of erroneous model data. The concrete characteristics of the configuration parameters are discussed in the context of the technical realization in the A3F (see section 10.3).

10.1.3 Excerpt of the Omni Model

This process step uses the information of the Omni Model. The main focus is on the System Model, the Integration Model, and the Test Model, which is the origin of the generated test cases. Independent of the applied execution mechanics of the ATE, especially the links between the System Model and the Test Model represent the central information. Information coming from other modeling domains of the Integrated Model Basis is not used for essential aspects in this process step. It is only conceivable that the linked information is incorporated into the generated execution logs, which can simplify subsequent evaluation and debugging because more contextual information is available. However, this requires a very disciplined creation and maintenance of those relationships, discussed in the concluding section of this chapter.

10.2 The Abstract Test Execution Approach

As already mentioned, the challenge of Abstract Test Execution in the context of model-based software development is that the completeness of the information cannot usually be assumed due to the intention of early applicability. Depending on the development state of the System Model, different types of information are available in the model and can therefore be evaluated for execution at this level.

In conventional software development, on the other hand, functionalities specified completely in the code are checked by test cases during execution. Depending on the integration level, either the respective code fragment or the entire system is executed under realistic conditions and subjected to the stimuli of the respective test cases. These stimuli, which either elicit a response from the system or directly manipulate the internal system state (depending on the interaction capabilities with the system), represent either theoretically arbitrary inputs from the system or non-specified components of the control flow. Similarly, the components of a test case determine the success or failure of the test case by matching returns from the system or occupancies of internal data structures with the expected value, respectively. All these actions are based at the code-level on unique identifiers that are assigned to variables or instances.

For our approach on the model-level, in particular, if models of `Level 0` or `Level 1` are provided, the presence of these identifiers and the internal data structures do not necessarily apply. On the other hand, in particular, executability in the sense of a compilation to object code including subsequent execution on the target system is not possible without further efforts. Therefore, two different approaches of ATE are presented, which use different concepts to analyze the available System Model and test cases in an integrated way. On the one hand, for the `Level 0` and `Level 1` instances of the present System Model, a structural interpretation of the model information is implemented against the background of the Integrated Model Basis. On the other hand, a static data flow analysis of the model information extended by dynamic aspects is implemented to be able to evaluate test cases against `Level 2` System Models. Overall, the fusion of the information available in the Omni Model marks the basis for both variants of our ATE, illustrated in figure 10.2.

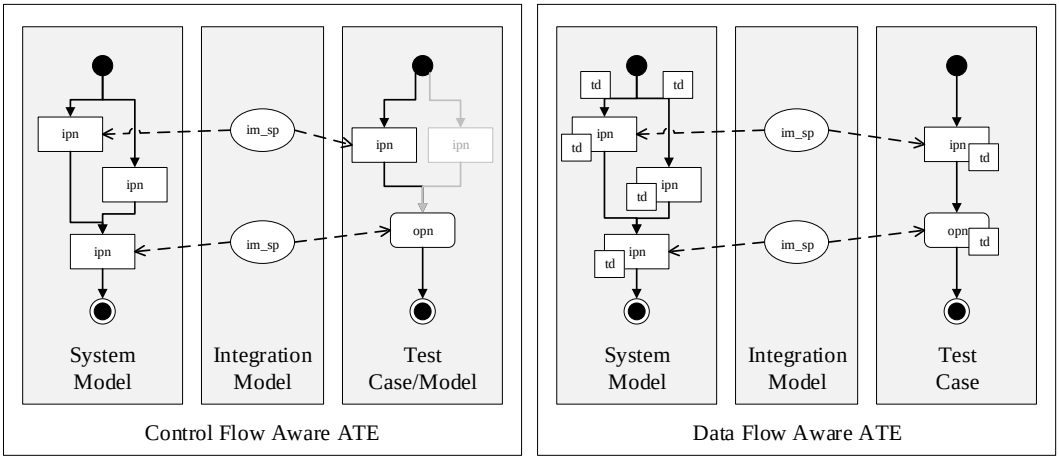


Figure 10.2: Omni Model foundation for the ATE approaches

The left side of the figure shows schematically on which information basis the *Control Flow Aware ATE* approach is built. The modeling domains *System*, *Integration*, and *Test* already mentioned in section 10.1.3 represent the basis. In particular, in the respective EGPP model elements of the *System* and *Test* domains are of particular importance, whereby a distinction is usually made between `EGPPInputNodes (ipn)` and `EGPPOutputNodes (opn)`. The former are visualized with sharp corners, whereas the latter are drawn with round corners. The underlying information is completed by the links between the two described modeling domains via the `IMSyncPoints (im_sp)` of the Integration Model and their connections to model elements of the respective domains.

On the right side, however, the scenario for the *Data Flow Aware ATE* approach can be seen. The information basis of the *Control Flow Aware ATE* approach is extended by detailed information. In this context, this detailed information is mainly code fragments, which are assigned to the said `ipn` and `opn` in the form of `EGPPTaggedData (td)`. In addition to the structural inter-dependencies specified via the Integration Model, further dependencies are specified in such a scenario through the cross-model use of identifiers in the context of `td`. This enables a comprehensive and code-oriented evaluation.

The two approaches are aimed at different phases of model-based development and are therefore not competing for approaches. However, in the course of the prototypical implementation of the two variants, some insights were gained that represent valuable information about the applicability of the approaches against the background of the MCSTLC.

First of all, we realized that for a meaningful evaluation of test cases against the present System Model by a Control Flow Aware Abstract Test Execution (CFA-ATE), a high degree of structural mappings has to be specified in the Integration Model. The mapping of all model elements of the Test Model or the Test Cases represents the desired goal, whereby a mapping of all `ipn` of the Test Model represents the minimum required information. This cannot be justified from the point of view of the modeling and model maintenance efforts and is contrary to the findings on the data basis of a Data Flow Aware Abstract Test Execution (DFA-ATE). Some experiments have shown that a rather small number of structural mapping over `im_sp` of the Integration Model lead to more meaningful results, in this case.

Furthermore, the collected findings of a CFA-ATE can hardly be transferred to concrete test verdicts. Above all, this is due to the abstract patterns analyzed in the context of this evaluation, which is based on purely structural information usually leads to vague statements. In the sense of a test case execution on structural information, no reliable and henceforth valid statement regarding the success or failure of a test case can be made from this. However, the information collected should not be underestimated concerning its value. For example, the results of the CFA-ATE can be used by the modeler to detect any *model smells* in the early stages and take appropriate countermeasures.

Based on these findings, the concept for CFA-ATE briefly presented in the following section is seen in terms of an Omni Model analysis for the early detection of modeling flaws, rather than a test execution mechanism. A reliable starting position for test activities of advanced development iterations usually can not be determined due to the problems mentioned. In the context of review activities, such an analysis can however efficiently uncover specification gaps or model smells (similar to *code smells*, but on the model-level). Due to the focus not being on the CFA-ATE, the following section just gives an overview of the CFA-ATE concept. Consequently, in the context of any process steps of the MCSTLC, the focus is on the DFA-ATE variant, since it is built on a resilient set of rules, which is discussed in detail.

10.2.1 Digression into a Control Flow-Aware ATE Approach

Building on model information of the Integrated Model Basis as schematically depicted in figure 10.2, CFA-ATE is intended to create a mechanism that provides information on the relationship between actual vs. target implementation in the very early phases of model-based development. The structural relationships that emerge from the Integration Model allow investigations to be carried out into the conformity of the two specifications at the model-level. For this purpose, a first concept was developed within the

scope of a master thesis that analyzes the execution path of the System Model caused by a test case and identifies issues in this context about the existing test case set. Originally, the idea was to perform an isomorphism check of the EGPP instances of the System and the Test Model against the background of the Integration Model data. However, this approach was discarded quickly due to the still open question regarding the complexity of this problem.

The concept of CFA-ATE is therefore based on a step-by-step examination of the graph structure, which is defined via the present Integrated Model Basis. Decisive at this point is the composition of the respective test case. The model elements contained in the test case in combination with the links between System and Test Model define the sections of the System Model, which are analyzed utilizing search algorithms. A possible connection between two model elements of the graph structure of the System Model is determined in a search iteration of such a section (see figure 10.3). During the analysis, all anomalies of the graph structure are recorded, which forms the basis for the final interpretation and processing of the test results. Against the background of an efficient concept, the search algorithm is improved by heuristic methods, where different properties of the graph structure are evaluated in advance and e.g. termination criteria are defined. After successful analysis, the collected information about the available graph structure is aggregated, processed, and transferred into a test report. This test report can be used in the early phases of development by different stakeholders in the model-based development process to improve the Omni Model to the identified problems. Details on the individual parts of the concept are based on content from [118] and are presented concisely in the following sections.

Challenges and Preprocessing

In the context of our search algorithmic approach for the CFA-ATE, several challenges exist. The first challenge is the scalability of the solution, which is primarily given by the properties of the model base at hand. The number of generated test cases plays a crucial role since a new search run is started by each test case. Likewise, the amount of links between the System and the Test Model is decisive, since this information guides the algorithm by defining sections of the iterative graph search. In general, the lack of knowledge about the properties of the System Model is an important aspect of scalability.

This lack of knowledge is a main aspect of the second challenge, the efficiency of the solution. In particular, due to the uncertainty about the completeness of the available model information, only a few assumptions can be made, which causes a reduction of the search space. Also, the definition of a criterion, which causes the termination of a search iteration, is difficult, since a static value always represents a compromise regarding efficiency and a value determined specifically for the System Model at hand requires a preliminary analysis, which again generates costs. If a search iteration has been aborted, another challenge of an efficient implementation is to find appropriate starting points for a resumption of the search or a reverse search.

To address some of these challenges, preprocessing of the model information is performed. To estimate the size of the System Model against the background of the search iterations to be performed, a user-defined number of random walks through the graph structure is performed. From this information, the termination criterion for a search iteration and at the same time the trigger for a reverse search is determined. The parameter k reflecting this purpose is defined as follows:

$$k = \frac{\gamma_{avg}}{2} \text{ with } \gamma_{avg} = \frac{\sum_{n=0}^{N} |RandomWalks|}{N+1}$$

The choice of the parameter k is a compromise between the search depth and the need for a backward search. It is important to note that the variant based on random walks does not produce deterministic results. If the expert wants to set a different value for the parameter k, this can be done via configuration parameters. In the case of the parameter being constant, the determinism of the results is given.

Furthermore, the System Model is checked for loops in advance. If loops are identified in the graph structure, the model elements they contain are marked appropriately so that they are handled accordingly during the search iterations of CFA-ATE. The same is done for the underlying Test Model. However, it is of particular interest whether the test case under consideration checks an iterative functionality of the System Model, i.e. several model elements of the Test Model are linked to the same model element from the System Model. This provides valuable information for the current analysis.

In addition to the information relevant to the algorithm, some metrics are determined during preprocessing and are listed in the generated test report. These include, for example, the ratio of the total number of model elements of the Test Model to the number of links to the System Model specified in the Integration Model. This is of particular interest in the case of searching for the root cause of a failing test case, while the mapping modeled in the Integration Model represents a possible source for such a test result. Together with other metrics for the existing Omni Model instance, valuable information is prepared that can be used by the expert afterward as a basis for troubleshooting.

Graph-Search Based Execution Concept

Based on the preprocessing, the concrete implementation of the algorithmic solution to CFA-ATE is discussed in this section. To clarify the initial situation of the execution concept, we refer to figure 10.3.

Similar to figure 10.2, this figure is split into the three modeling domains namely *System*, *Integration* and *Test*. On the right is the Test Model, the path showing a portion of a concrete test case. Here, v_T (an `EGPPInputNode (ipn)` of the Test Model) represents the node in the test flow under partial analysis. $pre(v_T)$ is the starting point of the past

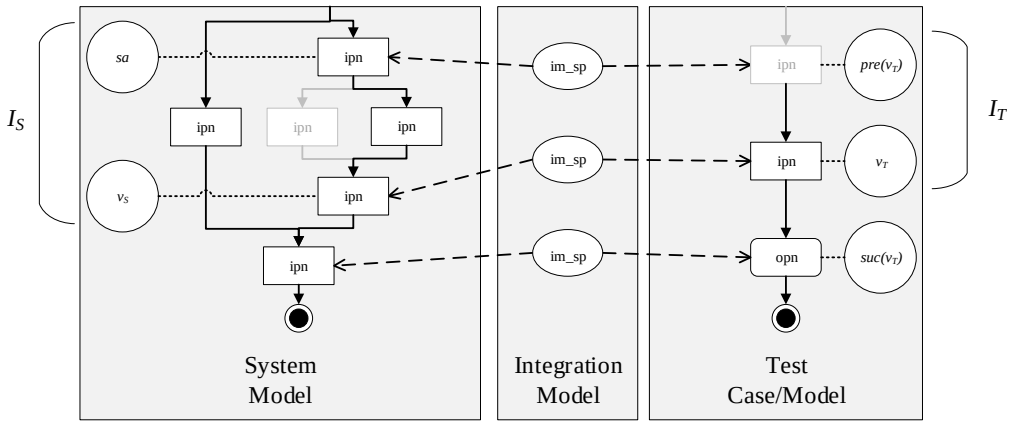


Figure 10.3: Schematic representation of the information basis of CFA-ATE

partial analysis, whereby the *searchAnchor sa* for the current partial analysis is defined via the links. The goal of a partial analysis is to find the next *sa* by starting a search in the System Model from the current *sa* towards v_S (an `EGPPInputNode (ipn)` of the System Model), which is determined by the Integration Model Information mapping v_T. The set of discovered paths between *sa* and v_S thus represents the possible execution paths for this interval.

This is done analogously for all pairs of consecutive model elements of a test case, where in the next step $suc(v_T)$ is in the focus, and v_S becomes the new *sa*. If one adds all partial results of a test case together after a successful analysis, one obtains all possible execution paths that can be evoked. In which way the partial analyses combine to the whole CFA-ATE is described by the following algorithm.

The implementation uses a combination of Breadth First Search (BFS), Depth First Search (DFS) and Backtracking, which enables efficient retrieval of the respective model element in the System Model (see algorithm 2). First of all, the *sa* for the initial model element of the test case is determined via the Integration Model ($im(v_{T\_init})$). Based on the *sa*, a BFS is performed taking into account the parameter *k* determined in advance. Furthermore, loops are taken into account by marking already considered model elements of the System Model and logging the corresponding execution paths. Finally, only one of the possible execution paths is taken as a representative result, but the information about the other possible execution paths is kept (see the grayed-out path on the left side of figure 10.3). The selection of a single representative ensures that an unambiguous result is obtained when the partial results are combined. If the search for the respective section is finished, it is distinguished whether a corresponding node v_S could be found or not. If this is not the case, a reverse search (backtracking) with the same parameters is performed. If no corresponding model element could be found here either, it is recorded at which point the search was aborted. If a model element could be found by one of the two mentioned searches, the corresponding execution path is recorded together with the model element. This is repeated until the analysis for the last model element of the test case has been performed and the next test case

can be examined.

Algorithm 2 Control Flow Aware Abstract Test Execution (based on [118])

```
sa = im(v_T_init)
v_S = null
while v_T ≠ null do
    sa = im(v_T)
    if sa ≠ null then
        v_S, executionPath ← SEARCH(sa, v_T, k)
        if v_S ≠ null then
            ADDRESULT(success, v_T, v_S, executionPath)
        else
            v_S, executionPath_rev ← REVERSESEARCH(sa, v_T, k)
            if v_S ≠ null then
                ADDRESULT(success_rev, v_T, v_S, executionPath_rev)
            else
                ADDRESULT(failed, v_T, v_S, ∅)
    v_T ← suc(v_T)
```

Graph-Search Result Interpretation

After successful preprocessing and execution of the iterative search algorithm, all information is available for the creation of the test reports. The following information per test case is derived:

- A set of model element pairs representing the corresponding elements on the part of the System and Test Model
- A possible execution path through the System Model caused by the test case composed of a set of partial execution paths for the examined intervals
- Information about the course of the analysis, e.g. reasons for aborting a search iteration
- Some metrics on the underlying models (see section 10.2.1)

Since it is difficult for the expert to process the collected raw information, an interpretation is required. In particular, the set of corresponding model elements in conjunction with the determined execution paths is therefore considered, allowing the determination of a set of possible findings. I_T represents the set of model elements of the Test Model contained in a considered interval. I_S, represents the set of System Model elements for the same interval (see figure 10.3). Based on this metric, the following interpretations are considered, determined by the structural modeling capabilities of the EGPPMM and the integrated analysis of the System and Test Model elements:

1. **System Artifacts missing:** If $|I_T| > |I_S|$ holds, i.e. in the considered interval the number of model elements of the Test Model is greater than the one of the System

Model, this indicates a specification gap in the System Model. Alternatively, it may be the case that the connections in the Integration Model are no longer up to date.

2. **Case Distinction not covered:** If more than one path is determined during the search for the corresponding model element of the System Model, this can be an indicator for a missing test case. If there is no other test case in the test suite that covers the other branch of the different cases, the System Model is not completely covered by the test suite and/or there is a weakness in the test case under consideration.
3. **System Loop not covered:** This case represents a special case of the previous case *Case Distinction not covered*. However, the discovered case distinction is part of a loop construct of the System Model, which is not taken into account by the test case. Formally such loops are detected based on the marked nodes, which are determined in the preprocessing step of the System Model.
4. **System loop missing or not traversed:** This variant represents a modification of the interpretation *Case Distinction not covered*, where according to the test case similar functionality is repeatedly checked, but not implemented on the part of the System Model in the form of a loop construct.
5. **Test Model Inaccuracy:** This interpretation is recorded if $|I_S|$ is significantly larger than $|I_T|$. Depending on the application context, this may indicate that the test case does not include some aspects. Likewise, this could be due to a more black-box type of test case, but this strongly depends on the ratio between $|I_S|$ and $|I_T|$ at hand. The threshold for the ratio of the sizes of these two sets represents a configuration parameter of the respective analysis, which defaults to factor 2.
6. **Integration Model Inaccuracy:** In this case, I_T and I_S each include a very large number of nodes, which indicates an incomplete specification of the connections in the Integration Model. However, it could be the case that further connections between nodes included in the interval do not make sense, which has to be investigated by the expert on a case-by-case basis. The threshold for the size of the two sets depends on the application context. Therefore it is realized as a configuration parameter, which defaults to 10.
7. **Merge:** The result is recorded if an until then unconsidered execution path of the System Model merges into the currently analyzed interval. This means the model element where the unconsidered execution path branched from the known execution path is not known to the analysis. Especially in the context of concrete troubleshooting based on the collected information, this can be an important indicator for possible side effects or emergent faults.
8. **Contradiction:** Based on the raw data, a contradiction can be detected in the models, in particular, if the forward search could not determine a valid execution path in the System Model, but only the backward search reveals a path to the corresponding model element.

All in all, the selection of interpretations clearly shows that only a few sound statements can be made due to the nature of the purely structural model data. A derivation of a concrete test verdict `PASS`, `INCONCLUSIVE`, or `FAIL` is therefore difficult to realize or is not usable in the test context concerning the reliability of the statement. Analyses that could not be completed successfully can be classified as `FAIL` with a high degree of certainty,

but a majority of the remaining test cases receives the test verdict `INCONCLUSIVE`. Regarding downstream processing in the context of the MCSTLC, the DFA-ATE approach is preferred, while the presented CFA-ATE approach offers valuable information to the modeler in the early stages of development. Overall, the CFA-ATE is therefore a promising base to support targeted debugging.

10.2.2 Overall Concept for Data Flow-Aware Abstract Test Execution

In contrast to the structure-based ATE, the CFA-ATE, the Data Flow Aware Abstract Test Execution (DFA-ATE) is introduced. As already shown in the right part of figure 10.2 (section 10.2), this concept combines the data specified in the System and Test Model with the structural relationships of the Integration Model. Since this approach requires additional model information, it is applied at a later stage than CFA-ATE, where detailed models are available.

The concept of DFA-ATE consists of the following parts: In the first processing step, *Omni Model-Based Path Merging*, the respective test case is merged with the corresponding System Model according to a set of rules, that are detailed in the respective section. During this process, the connections between the System Model and the test case stored in the Integration Model are evaluated, e.g. to obtain the correct entry point into the System Model. Furthermore, the merging is initially performed for a so-called *Segment*, describing the section of a test case between two verifying statements (see figure 10.4). Since this process is based on a static analysis, the result is a set of combined execution paths for the respective Segment. These paths represent the execution paths that are initially obtained without a detailed evaluation of the underlying information. In the next step (*Evaluation of Path Space*) the determined set of merged execution paths for the segment is examined concerning control and data flow issues. In particular, detected problems are put into context and recorded according to the different execution paths, being the basis for the subsequent calculation of a Test Verdict. These two processing steps are executed alternately until all Segments have been analyzed for the respective test case. After the iterative process of evaluation is completed, the next step (*Evaluation Result to Test Verdict Mapping*) is performed on the intermediate result. Here, the collected information on the identified execution paths is evaluated and, based on a set of rules, mapped to a Test Verdict. In the final processing step (*Result Selection and Test Report Generation*) a representative result is selected from the set of possible execution paths, their information, and derived test verdict. Afterward, the representative result is processed for different application contexts, such as machine processing in the context of MCSTLC or as feedback for the modeling expert, and finally converted into a suitable test report.

The concepts and illustrations presented throughout this chapter are based in part on [131], [83], and [147]. A portion of the running example from section 6.2 is further used to illustrate the concepts.

Excerpt of the Running Example

In contrast to the System Model components at lower integration levels, such as those used to illustrate the Test Case Management concepts in chapter 8, the ATE context makes use of the higher-level model parts of the CSM. Therefore, the left side of figure 10.4 shows the EGPP instance of the statemachine of the CSM system, which manages the activation of the whole system.

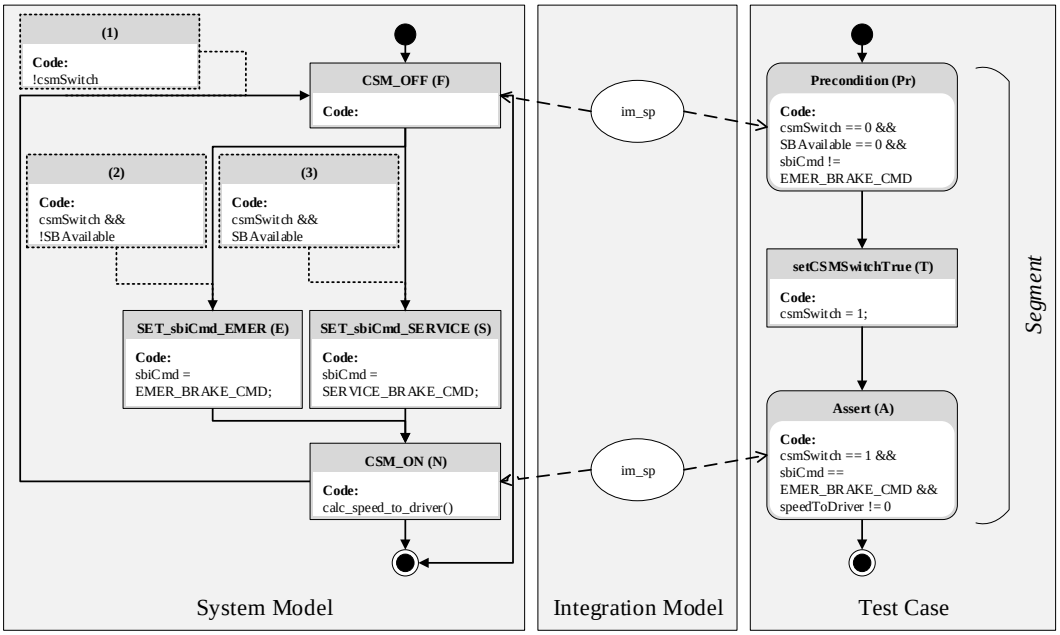


Figure 10.4: Omni Model excerpt for the CSM Running Example

Therefore, the System Model is divided into the different `EGPPInputNodes` (**F**, **E**, **S**, **N**) which may contain code fragments for the detailed specification of functionality and are visually represented as rectangles. The edges between the nodes define the control flow and may specify guards (see **1**, **2**, **3**). On the right side of the figure, an exemplary test case is shown, which was derived from a corresponding Test Model. This in turn is composed of `EGPPInputNodes`, which describe the test steps, and `EGPPOutputNodes`, which describe the verification points of the test case (see **Pr**, **T**, **A**). Both types of nodes contain code fragments describing the concrete characteristic. The `EGPPOutputNodes` are each shown with rounded corners throughout the rest of this section. The two model instances are related via the connections stored in the Integration Model using the `IMSyncPoints`. These represent the structural dependency between instances of the different Omni Model domains, in this case, System and Test. Further, the `EGPPOutputNodes` included in the test case determine the number of *Segments* to be analyzed. The illustrated test case includes one such Segment, which is drawn in at the right edge of the figure.

Apart from the visualization aspects, the test case verifies that the initially switched-off CSM system (`csmSwitch = 0`, `SBAvailable = 0`, `sbiCmd = SERVICE_BRAKE_CMD`, `speedToDriver = 0`), which is in a certain overall state (**F**), is started by the trigger

`csmSwitch = 1` and computes plausible initial values during its first cycle. In particular, the initial values are determined by functionality embedded in the `EGPPGraph` of **N**, i.e. specified at a different integration level.

10.2.3 Omni Model-Based Path Merging

The first processing step deals with the determination of possible execution scenarios of the considered test case against the present System Model, which leads to a set of possible execution paths. As already mentioned, the merging is always performed per segment before an evaluation of the determined paths for the segment is performed.

To perform the merging of the model information in a meaningful way, the start and end points of the analysis have to be determined in advance. Due to the nature of the EGPP instances (see section 7.2.1), corresponding start and endpoints are given. However, without further constraints, the set of model elements that need to be examined would be very large, which is due to the ability of EGPP Models to nest instances across multiple levels. By connecting the first and last `EGPPOutputNodes` of the test case via the Integration Model, there is an opportunity to explicitly specify these points and thus reduce the set of model elements that have to be considered. The specification of a starting point is optional, but the specification of an endpoint is necessary since the endpoint appropriate for the test case is difficult to be determined automatically. Furthermore, this endpoint is a critical point for control and data flow analysis in the context of DFA-ATE.

Based on this information, the merging of the model elements can be started. Therefore, the model elements contained in the test case are iteratively integrated into the System Model. The simplest approach would determine and evaluate all possible permutations resulting in many variants that contradict the EGPP metamodel or are not appropriate in the context. To counteract the state explosion that occurs at this point, the following rules are specified, which produce a reduced set of execution paths.

1. The order of the model elements of the System Model and the test case is preserved.
2. The combination of an edge and a node of the System Model represents an inseparable entity (see definition 34) which cannot be split by model elements of the test case.
3. The combination of an `EGPPOutputNode` (Verifying Node) and the subsequent `EGPPInputNodes` (manipulating nodes) of a segment of the test case represent an inseparable unit

The first rule is largely self-explanatory, since the intervention in the order of the model elements of the two EGPP models (system and test) represents a change in the modeled behavior, and therefore the results derived from this would lose any validity.

The second rule is based on the semantics of the EGPP metamodel. Here, the edge

represents the transition between two system states, whereby this can be protected by a guard (verifying instruction). Traversing such a guard is usually achieved during testing by stimuli on the part of the test case, which corresponds to one or more `EGPPInputNodes` that mimic the desired effect before the edge under consideration and its guard are evaluated. This basic idea of testing and the given semantics of the EGPP metamodel provides the rationale for this rule.

The third rule is the counterpart of the second rule on the part of the test case. It is assumed that starting from an unmodified system state confirmed by an `EGPPOutputNode` (verifying node) of the test case, a set of stimuli (`EGPPInputNodes`) of the test case are processed before the system responds to them. Further, due to the abstract approach of DFA-ATE, a temporal consideration is not performed. That is, if more than one guard-protected node of the System Model is included in the currently considered segment, the test case may fail by mistake. The reason is that the two model parts have for themselves a fixed order, however, with the unification within a segment, no information about the intended order is available. Therefore, in order not to exclude any valid variant of an execution path the concept of *Overassignment* of variables within a segment is implemented.

The *Overassignment* allows variables that are part of the system state to have several valid assignments for the observation context of a segment. This ensures that the stimuli given by the test case are retained until they are needed, e.g., to fulfill a guard. If such an Overassignment is utilized to lead the test case to success, this is taken into account accordingly during the derivation of a Test Verdict (see section 10.2.5).

To illustrate the rules shown, figure 10.5 shows the total set of determining execution paths P_{lim}.

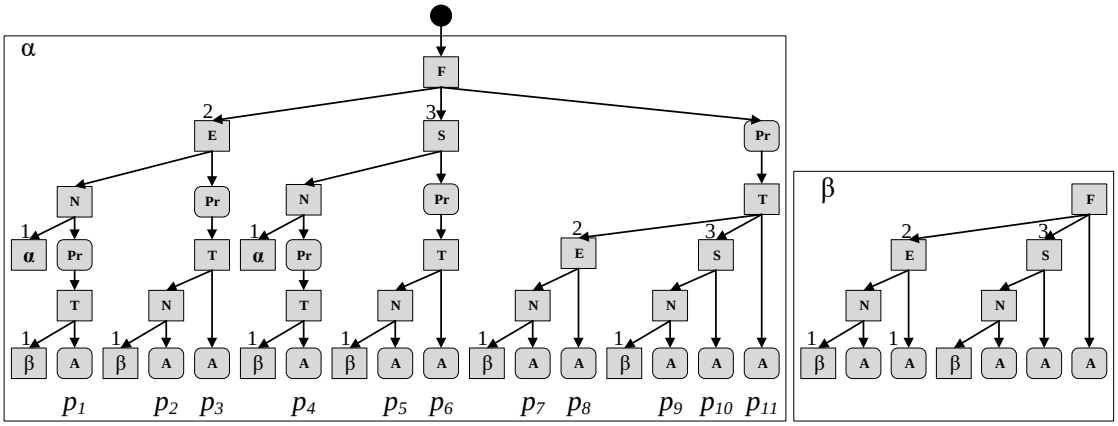


Figure 10.5: Execution paths for the Running Example from figure 10.4

In this figure, the shaping and identifiers of the model elements from figure 10.4 have been adopted. That is, the first-named node at the top of α represents the `EGPPInputNode` **F**. The possible paths from the root node to a leaf of the tree reflect the result determined according to the rules. Since the given System Model contains loops, the

structure is repeated at different parts of the graph, hence by α or β the respective subtree is applied. The identifiers $p_1 - p_{11}$ shown at the bottom of the figure represent the different paths, each ending with the `EGPPOutputNode` **A**, representing the end of the considered segment. The paths of β ending with **A** are already included on the right side of α, such that $p_7 - p_{11}$ cover the possible cases of this recurring part. At this point, no reduction of the path space can be achieved by the information of the Integration Model, since only the two nodes **F** and **N** of the System Model are linked.

10.2.4 Evaluation of Path Space

As soon as the paths for a segment are determined, the evaluation of the path space P_{lim} starts with the data flow and the control flow. To ensure that the code fragments contained in a path can be evaluated correctly, the associated control flow makes sense and the specified endpoint of the System Model is reached.

In the first step, the data flow analysis is executed per path, where a wide variety of faults can occur. The set of possible faults is defined as $D := \{d_1, d_2, d_3, d_4, d_5, d_6\}$, whereby the possible fault types are represented in the following list.

- d_1 The code fragment of the considered node could not be solved (e.g. Verification Node originated in the Test Model)
- d_2 The guard of the considered transition could not be solved
- d_3 The code fragment of a node or a transition contains several undeclared or uninitialized variable(s)
- d_4 The respective endpoint could not be determined
- d_5 The guard of the considered transition contains a/several time-dependent variable(s) (e.g. guard conditions regarding the system time)
- d_6 The guard of the considered transition is solved by an/several over assigned variable(s)

By mixing concepts from the System Model and the Test Model, the subsequent analysis does not distinguish between the domains of the model elements, but only between Variable Verifying Instructions (VVIs) and Variable Modifying Instructions (VMIs), i.e., model elements that check the system state VVI and those that change the system state VMI. For example, a VVI is considered non-solvable unless the boolean expression can be evaluated to `true` in the current system state. The analysis of such a segment can be represented in pseudo-code, as shown in its basic form in [147] and illustrated in algorithm section 10.2.4 in an extended version.

The functions applied during analysis are presented in a more structured way in algorithm 3. The method EVALSEGMENT(s), called for one segment s of the test case at a time, represents the starting point. Here, either the variable assignments are updated for the model elements to be evaluated (see STOREVALUESOFELEMENT) or a boolean expression is evaluated based on the current variable assignment (see VERIFYELEMENT). Finally, the PERSISTLASTSTOREDVARIABLEVALUES function en-

sures that the overassignment mechanism implemented within the segment does not affect subsequent segments by resolving overassigned variables. As a counterpart to storing these values, the different assignments are taken into account when evaluating instructions (see GETPERMUTATEDVARIABLEASSIGNMENTS(I)). Across all functions, the REGISTERFAULT(d_x, e) logs the corresponding faults.

Algorithm 3 Algorithm for the evaluation of an ATE segment s

```
function EVALSEGMENT(s)
    for e ∈ GETELEMENTS(s) do
        if INSTANCEOF(GETINST(e), VMI) then STOREVALUESOFELEMENT(e)
        else if INSTANCEOF(GETINST(e), VVI) then VERIFYELEMENT(e)
    PERSISTLASTSTOREDVARIABLEVALUES(s)

function VERIFYELEMENT(e)
    if NEWFAULTSREGISTERED(CHECKPRECONDITIONS(e)) then return
    if INSTANCEOF(e, EGPPNode) then
        if not VERIFYINST(GETINST(e)) xor ISOVERASSIGNED(GETINST(e)) then
            REGISTERFAULT(d_1, e)
    else if INSTANCEOF(e, EGPPTransition) then
        if VERIFYINST(GETINST(e)) then
            if ISOVERASSIGNED(GETINST(e)) then
                REGISTERFAULT(d_6, e)
        else
            REGISTERFAULT(d_2, e)

function CHECKPRECONDITIONS(e)
    for v ∈ GETVARIABLES(GETINST(e)) do
        if ISTIMEDEPENDENTVARIABLE(v) then REGISTERFAULT(d_5, e)
        if SIZE(GETSTOREDVALUES(v)) = 0 then REGISTERFAULT(d_3, e)

function ISOVERASSIGNED(i)
    for v ∈ GETVARIABLES(i) do
        if SIZE(GETSTOREDVALUES(v)) > 1 then return true
    return false

function STOREVALUESOFELEMENT(e)
    if INSTANCEOF(e, EGPPGraph) then STORE(GETINSTOFSTUB(e))
    if INSTANCEOF(e, EGPPNode) then STORE(GETINST(e))

function VERIFYINST(i)
    for va ∈ GETPERMUTATEDVARIABLEASSIGNMENTS(i) do
        if EVAL(i,va) then return true
    return false
```

A special feature of the extended variant of the pseudo-code can be seen in the method STOREVALUESOFELEMENT(e), since the *stub* concept available in the EGPP context, is addressed here (see section 7.2.2). The stub graph, which is connected to an `EGPPGraph`, may contain black box information about already executed tests on lower integration levels. *Black Box Information* in this context means that the *Stub* contains information about the outputs or variable assignments based on a set of input variables. Provided that the required input variables of the stub coincide with the current system state before the evaluation of the relevant `EGPPGraph`, the new system state can be derived directly from the outputs stored in the stub. Unless there is a match in the inputs, the EGPP model contained in the `EGPPGraph` has to be analyzed, but this requires significantly more effort. The stub concept can be seen as a performance improvement in this context if extensive testing has been done especially on low integration levels and therefore comprehensive information is available in the stub models. In combination with the data flow analysis of the paths, a control flow analysis is performed, which in particular examines whether the path under consideration could be successfully terminated at the specified endpoint. This type of control flow analysis relies on the information of the Integration Model, where the structural relationships are stored. Analogous to the set of different fault types for data flow analysis, a set $C := \{c_1, c_2, c_3, c_4\}$ describing the different control flow faults was defined.

- c_1 All verifying instructions of the path are fulfilled and the last verification point is solved by the instructions of one of the end nodes of the System Model
- c_2 All verifying instructions of the path are fulfilled and the last verification point could not be solved using the instructions of one of the end nodes of the System Model
- c_3 At least one verification point of the path could not be fulfilled, but one of the end nodes of the System Model is part of the path
- c_4 At least one verification point of the path could not be fulfilled, but no end node of the System Model is part of the path

We first distinguish whether the present path in the data flow analysis could satisfy all VVIs (c_1 and c_2) or not (c_3 and c_4). In the first case, it is further distinguished whether the termination of the path is given in an endpoint specified by the Integration Model (c_1) or whether another node is the termination (c_2). In the second case, it is only distinguished whether one of the specified end nodes is part of the failed path execution (c_3) or not (c_4).

Overall, a set of faults is recorded for each execution path, from which a set $O_S := \{R_1, \ldots, R_{|P_{lim}|}\}$ is created for the entire segment, where $R := D_M \cup C_P$. Here, D_M represents the set of data flow results, where an element of this set always is a pair $D \times M$, with M being the set of model elements (nodes and edges) of the merged EGPP model. C_P, in contrast, represents the set of control flow results, with each element being a pair $C \times p_x$.

For our considered running example, which was already introduced in section 10.2.3, the following set $O_S := \{R_1, \ldots, R_{11}\}$ results according to the algorithm.

$$
\begin{aligned}
R_1 &= \{(d_2, 2), (d_1, Pr), (c_3, p_1)\} \\
R_2 &= \{(d_2, 2), (d_1, Pr), (c_3, p_2)\} \\
R_3 &= \{(d_2, 2), (d_1, A), (c_4, p_3)\} \\
R_4 &= \{(d_2, 3), (d_1, A), (c_3, p_4)\} \\
R_5 &= \{(d_2, 3), (d_1, A), (c_3, p_5)\} \\
R_6 &= \{(d_2, 3), (d_1, A), (c_4, p_6)\} \\
R_7 &= \{(c_1, p_7)\} \\
R_8 &= \{(d_1, A), (c_3, p_8)\} \\
R_9 &= \{(d_2, 3), (c_3, p_9)\} \\
R_{10} &= \{(d_2, 3), (d_1, A), (c_4, p_{10})\} \\
R_{11} &= \{(d_1, A), (c_4, p_{11})\}
\end{aligned}
$$

The sets R_1 - R_{11} show the faults for the respective paths p_1 - p_{11}. For example, R_1 includes three elements, where the first element describes that the guard `csmSwitch && !SBAvailable` of the edge **2** could not be satisfied. The second element describes that furthermore the condition `csmSwitch == 0 && SBAvailable == 0 && sbiCmd != EMER_BRAKE_CMD` of the node **Pr** could not be fulfilled. Finally, the last element of the result set describes that at least one condition of a node could not be satisfied, but one of the specified end nodes is part of the path. The results of the other paths are to be interpreted analogously. All in all, this information provides the basis for the derivation of a Test Verdict, presented in the following section.

10.2.5 Evaluation Result to Test Verdict Mapping

In conventional test frameworks, a so-called Test Verdict is derived for each test case. This is primarily used to be able to classify the execution of the test cases without having to look at the detailed information about the test execution. Usually, a distinction is made at this point between `PASSED`, `INCONCLUSIVE`, `FAILED`, and `ERROR` [82]. The last Test Verdict maps the state indicating that an error occurred on the part of the execution framework. In the further explanations of the concept, an idealized execution is always assumed, which is why the test verdict `ERROR` is not considered anymore.

In the context of DFA-ATE, however, this set of different Test Verdicts cannot be used to express the difference whether the cause of failure (Test Verdict ≠ `PASSED`) is due to our concept of ATE or is located in the modeling of the system or test. To be able to represent this difference in the Test Verdicts, `PROBABLY PASSED` is added to the set of Test Verdicts mentioned above. Specifically, this Test Verdict is used if during our data flow analysis it was determined that the execution path either contains a time-dependent variable (d_5), or the overassignment concept was used to further pass the execution path (d_6), but no other faults were logged. To bring all remaining test verdicts in line with our specified fault types and analysis results, the following definition is given to derive the Test Verdict from the collected analysis results [147]. For the Test Verdicts a shortened notation is used in the further course, i.e. `pa` for `PASSED`, `pp` for `PROBABLY PASSED`, `in` for `INCONCLUSIVE` and `fa` for `FAILED`.

$$M : O_S \to V \begin{cases} pa, & \text{if } \exists (f,h) \in R.\ f = c_1 \wedge |R| = 1 \\ pp, & \text{if } \exists (f,h) \in R.\ f = d_i \text{ such that } i \in \{5,6\} \wedge \\ & \quad \forall (f,h) \in R.\ f \neq d_j \text{ such that } j \in \{1,2,3,4\} \\ in, & \text{if } \exists (f,h) \in R.\ f = d_i \text{ such that } i \in \{3,4\} \wedge \\ & \quad \forall (f,h) \in R.\ f \neq d_j \text{ such that } j \in \{1,2,5,6\} \\ fa, & \text{otherwise} \end{cases}$$

Accordingly, the verdict `pa` is determined by a data flow analysis without any faults recorded, whereby the control flow analysis has to be terminated at the specified endpoint (c_1). The verdict `in`, on the other hand, is determined if only faults of the classes d_3 and d_4 were determined during the data flow analysis. All result sets that do not match any of the criteria of the already explained Test Verdicts `pa`, `pp`, or `in` lead to a Test Verdict `fa`. As mentioned above, the classical Test Verdicts are thus assigned the commonly used semantics, and the specifics of our approach are mapped into the novel Test Verdict `pp`. Overall, however, a pessimistic form of the derivation of the Test Verdicts is implemented at this point, which means that an execution path is assigned to a worse Test Verdict in case of any doubts. The idea behind this is that *false positives* often remain undetected in the test context, but can cause considerable damage, whereas *false negatives* may require an unnecessary manual review, which is relatively uncritical.

As in the other sections, this process step is again illustrated by our running example. Specifically, table 10.1 shows the Test Verdicts determined from the sets $R_1 - R_{11}$.

R_x	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9	R_{10}	R_{11}
$M(R_x)$	fa	fa	fa	fa	fa	fa	pa	fa	fa	fa	fa

Table 10.1: Derived Test Verdicts for the Running Example

Only the set R_7 of the execution path p_7 is evaluated for the Test Verdict `pa`, whereas all other sets are assigned the Test Verdict `fa`. The mapped test verdicts thus represent all theoretically possible results of an execution of the considered test case.

10.2.6 Result Selection and Test Report Generation

Continuing with an analyzed and classified set of execution paths is available for the test case under consideration, the last step of the DFA-ATE is to select a representative result, which is listed in the test report. For this purpose, an order is defined on the presented test verdicts. This serves essentially to create a clear decision basis for the mentioned selection process. Based on the argumentation for the introduction of the novel test verdict `PROBABLY PASSED`, $pa >_v pp >_v in >_v fa$ applies in the further course of the work, where `pa` represents the best and *fa* the worst result.

On this basis, in contrast to the pessimistic derivation of a test verdict from analysis information, an optimistic selection of the representative is made, which means that the best test verdict also becomes the representative for the test case. Based on the definitions introduced in advance, the following set of rules results for the selection process.

$$\Sigma : O \rightarrow O_{best} := \{\arg\max_{>_v}(M(R_i)) \wedge \arg\min(|p_i|) : \\ R_i \in O \text{ , } p_i \text{ being the respective path for } R_i \text{ with } i \in \{1, \ldots, |P_{lim}|\}\}$$

The formula describes that the best result is determined from all analysis results O for the paths by first filtering those paths with the best Test Verdict. If multiple paths are leading to the same Test Verdict, the execution path that includes fewer model elements is preferred. If these two constraints do not lead to an unambiguous result, a path is selected at random, and the application of this selection criterion is indicated to the expert in the test report.

Abstract Test Execution Report

After the successful determination of a representative test result per test case, the ATE is complete, i.e. all artifacts are generated. In particular, the *Abstract Test Execution Report* designed for modeling experts can be derived. Listing 10.1 shows the essential excerpts of such a report for the Running Example.

```
1  ================================================================================
2  Test Case: TC005_CSM-Statemachine_{B7BA8801-ECC1-4fc2-B992-874737C5C528}
3  ================================================================================
4  System Under Test: CSM-Statemachine
5  Test Result: PASS
6  Test Steps:
7  --------------------------------------------------------------------------------
8  1. Precondition ["csmSwitch == 0 && SBAvailable == 0 &&
9  sbiCmd != EMER_BRAKE_CMD"]
10 2. setCSMSwitchTrue ["csmSwitch = 1;"]
11 6. Assert ["csmSwitch == 1 && sbiCmd == EMER_BRAKE_CMD &&
12 speedToDriver != 0"]
13 --------------------------------------------------------------------------------
14 System Steps:
15 --------------------------------------------------------------------------------
16 0. CSM_OFF [""]
17 3. ["csmSwitch && !SBAvailable"]
18 4. SET_sbiCmd_EMER ["sbiCMD = EMER_BRAKE_CMD;"]
19 5. CSM_ON ["calc_speed_to_driver();"]
20 --------------------------------------------------------------------------------
21 Execution Log:
22 --------------------------------------------------------------------------------
23
24 INFO: We passed all VPs and last VP was solved with the specified Endpoint!
25 --------------------------------------------------------------------------------
```

Listing 10.1: Test Report for Running Example CSM statemachine

The test report for a test case is divided into different sections. The upper section of the report contains general information about the input artifacts and the determined result. It is shown which test case was derived from the Test Model and which System Model was considered. The determined Test Verdict is directly visible. If detailed information is desired, in the middle section of the report the run through model elements of the test case (lines 8-11), as well as the model elements of the system are presented (lines 15-19). In particular, the overall execution sequence is shown transparently by the overarching numbering presented. In combination with the execution steps, the lower part of the report (lines 21-25) formulates the collected data flow and control flow errors in natural language and gives a reference to the concerned model elements. In this case, there were no faults detected and therefore no such content is displayed. In contrast, listing 13.4 gives an example of how detected faults are illustrated in the execution log of a test report.

10.3 Technical Realization within A3F

As with the two MCSTLC processing steps presented above, a prototypical implementation in the A3F is realized for the two *Abstract Test Execution* variants. The basis is the explained concepts, which are encapsulated in a configurable analysis of the A3F. To get an overview of the configuration possibilities, the available parameters of the `egpp_execute` analysis are listed and briefly explained in table 10.2.

Table 10.2: Configuration parameters for `egpp_execute` analysis

Parameter	Description
`integrationmodel`	Determines the Integration Model which manages the Omni Model around the `systemmodel` and the `testmodels`
`systemmodel`	The System Model which represents the basis for the application of mutations
`testmodels`	The Scoped Test Models from which the `testcases` are derived
`testcases`	Determines the set of test cases which are processed by the mutation analysis
`executor`	Determines the executor, for the Abstract Test Execution. Possible values are `controlflow` and `dataflow`
`executorconfig`	Determines a set of configuration parameters for the respective `executor`

A part of the configuration parameters concerns the necessary model information, which is either taken directly from the Integrated Model Basis or represents results of upstream analyses. In particular, the parameters `integrationmodel`, `systemmodel`, and `testmodels` describe the model artifacts which are the basis for the execution. The parameter `testcases` returns the pre-generated test cases that have emerged from the

mentioned `testmodels`. In addition to these input models, the parameter `executor` can be used to select the desired variant of the execution mechanics. On the other hand, the parameter `executorconfig` can be utilized to pass a set of configuration parameters depending on the choice of the execution mechanism. These parameters allow an expert to optimize the execution mechanism for the problem size.

A snippet of a possible analysis chain where the `egpp_execute` analysis is integrated can be seen in figure 10.6.

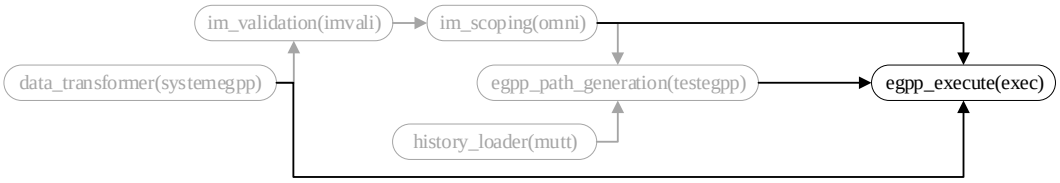


Figure 10.6: Analyses dependency graph for the `egpp_excute` analysis

In particular, this graph shows the dependencies concerning upstream analyses of the A3F. These are shown in gray, with the arrows describing the flow of information between the analyses. For example, the two parameters `integrationmodel` and `testmodels` represent results of the displayed `im_scoping` analysis. The System Model is derived from the `data_transformer` analysis. The test cases, which are determined by the `egpp_path_generation` analysis, complete the model information. A concrete configuration for the `egpp_execute` analysis can be seen in listing 10.2.

```
1  <configurations of preceding analyses>
2
3  analysis egpp_execute(exec) {
4    integrationmodel=
5      "im_scoping|omni|IMScopingResult|getFilteredIMModel";
6    systemmodel=
7      "data_transformer|systemegpp|DataTransformationResult|getOutputElements";
8    testmodels=
9      "im_scoping|omni|IMScopingResult|getExtractedMBTModel";
10   testcases=
11     "egpp_path_generation|testegpp|EGPPPathGenerationResult|getTestCases";
12   executor="dataflow";
13   executorconfig="10,10";
14 }
```

Listing 10.2: Example configuration for the `egpp_excute` analysis

After successful execution of this analysis configuration, a set of reports is available for execution of the individual test cases. On the one hand, these reports provide the expert with information about the execution sequence of the individual test cases, and on the other hand, they can form the basis for more advanced mechanisms such as mutation analysis (see chapter 11). Applying the shown configuration to the Running Example used for illustration leads to the result artifacts, some of which were already shown in the concept part of this chapter. Further examples of this kind are provided in the evaluation chapter by the case studies.

10.4 Related Work

In addition to the presented approaches to ATE, many other concepts exist for the realization of execution or evaluation of development artifacts of the model-level. The spectrum ranges from formal approaches that verify the existing models to approaches that simulate the models. Further, the used modeling languages span an extensive spectrum, whereby altogether the UML and its variants represent a significant amount. For example, Ciccozzi et al. have carried out an extensive analysis that presents a wide variety of approaches to executing UML Models and categorizes these approaches [43]. In particular, the categorization into *interpretive* and *translational* is made, while our approaches can in some sense be assigned to both groups. *Translational* about the M2MTs performed towards the EGPP representation and *interpretative* concerning the model interpreter specified in the context of our DFA-ATE.

However, to discuss approaches beyond UML, the presented related work is subdivided into *Model Translation and Model Interpretation* and *Formal Model Verification*.

Model Translation and Model Interpretation

As noted in the systematic literature review by Ciccozzi et al. [43], there is an imbalance between *interpretive* and *translational* approaches. While the majority of existing approaches choose to transform the original model artifacts to a modeling language having an execution engine, there are only a few approaches that interpret the model information without further preprocessing. When translating the model information, different target languages can again be chosen, and few engines are widely used. For example, the MoMuT::UML framework uses Object Oriented Action System (OOAS) as the target model, which is then executed by the corresponding engine, and test cases are examined in the same context [115]. A generalization towards a generic model execution engine supporting different modeling variants is shown, e.g. by Kirshin et al. [112]. Other approaches do not use a modeling language as a target language, but rather established programming languages. A well-known representative of this is the established development framework MatLab/Simulink, whereby C is the target language for the simulation [48].

This contrasts with the interpreter-based approaches to model execution. Common interpretation engines are mostly based on UML, for example, the *Moka* framework based on Eclipse Papyrus [55]. Whereas other approaches use the fUML engine to interpret the model information [124][125][164]. Moreover, there are interpreters developed for specific use cases, e.g. Crane et al. who have implemented their UML Virtual Machine [46].

However, compared to our ATE approach, all related approaches known to us lack flexibility for the chosen modeling language for the specification of SUT. Due to this fact, our approach, which performs a transformation towards our EGPP representation and

carries out an interpretation of the generated model instances, can convince by its flexibility. Another aspect emerges especially in comparison with *translational* approaches, which use established programming languages as target artifacts. In case of missing detailed information in the available models, no executability can be achieved by such approaches, whereas ATE can cope with models of different development stages. This feature allows to execute models at earlier stages of the development and to draw insights from the test cases accordingly mitigating early errors.

Formal Model Verification

In contrast to testing, formal verification compares the existing model against a formal behavior specification. For this purpose, the requirements are often formalized, and subsequently evaluated against the possible system states [27]. In the context of our approach, the components of the test cases are matched against the possible system states, but utilizing structural and data flow-based relationships. Furthermore, our semi-formal specification does not have to be created by the user but is derived automatically from the Test Model. An examination of the system is accomplished by considering all possible execution paths, implemented by various approaches in the context of *Symbolic Execution* [110][109][130]. For this purpose, however, complete specifications are advantageous to obtain a useful result. Further development of the Symbolic Execution is the *Abstract Execution*, which continues the evaluation with placeholders in case of incomplete specifications [161].

Compared to the model checker concepts our basic concept offers the advantage that only a fraction of the theoretically possible execution paths have to be considered since the subsystem in question is known via the structural relationships. Furthermore, the complex evaluation algorithm is transparent to the user, who therefore merely specifies the information in the familiar modeling environment and then determines the structural relationships. Expert knowledge of formalization approaches is therefore hardly necessary, which often makes access to formal approaches difficult.

10.5 Conclusions and Outlook

Within the scope of this chapter, approaches to Abstract Test Execution were presented, which are mainly applied in the early phases of the MDSD. On the one hand, a concept was presented that mainly evaluates structural relationships of the Integrated Model Basis and implements a type of test execution that can be seen as an approach for modeling support. Likewise, a concept was shown, which is intended for subsequent phases of the MDSD. For this purpose, this concept implements a model interpreter based on the Integrated Model Basis and the existing data flow information, which already enables ATE at this stage. In response to the question posed at the beginning

How would an automated mechanism for the verification of test cases based on (probably incomplete) models of the SUT look like, reducing the emerging test complexity?

the following answer can therefore be given.

The basis of both approaches is provided by the EGPP representation of the original model artifacts. In connection with the Integration Model and the extensive definition of operational semantics, a solid basis for execution at the model-level is given. Through the control flow-based approach for the execution of test cases on this model basis, a statement about the conformity of the model artifacts can be made in the early phases of the development. Likewise, any modeling weaknesses can be identified in very early phases, which can be corrected accordingly with reduced effort and in the context of lower complexity. The data flow-based approach for executing the model information can additionally be used to check the conformity of the specified details, taking into account the structural relationships. By automating both approaches as far as possible, the complexity of the problem is hidden from the user, who can only influence the execution via configuration parameters. Furthermore, it was paid attention to the fact that the produced results come as close as possible to a conventional test report, to be able to convert the application scenarios known from the code-level with small expenditure on the model-level and to make a smooth transition possible between the development contexts. Overall, the focus was on a clean integration with the already presented process steps of the MCSTLC, whereby the emergent complexity remains manageable.

Besides the positive aspects of the approach, there are some challenges to be addressed explicitly. The first to be mentioned is the Omni Model, which plays the most sensitive role in the context of ATE. Since for both concepts of our ATE the structural relationships of the Integration Model play a central role, the correct modeling and updating of the underlying information are of high relevance. Incorrect or outdated model information can, in the worst case, create a false picture to the test case set, which makes the application of the approach dangerous. Another aspect of our approach is the included model transformation step for the System as well as the Test Models. If the pre-specified model transformations create an erroneous model base in the EGPP environment, the obtained ATE results are not usable. Similarly, the model transformations create an additional indirection to the models and the modeling language used to create the particular system. This is a disadvantage for localizing the cause of the error unless the model transformation is implemented bidirectionally, but this is again related to the one-time initial effort. The last challenge related to the presented ATE concepts is the scalability of the approach. Both concepts scale as long as the Integrated Model Basis provides sufficient structural relationships between System and Test Model, as well as the modeling, is aligned. This means especially for the data flow-based ATE that the model interpreter does not have to consider several hierarchy levels of the System Model when executing test cases of a certain integration level. In this case, the path space for the respective test cases would become enormously large, which increases the effort exponentially. Provided that the causes addressed here are taken into account in the application context, the positive aspects outweigh the negative ones.

Further work in the context of the presented concepts includes, for example, a more specific preprocessing of the available model information. This could ensure or at least improve the aforementioned scalability challenges in special cases. A further improvement of the scalability could be achieved by an additional analysis of the already available Test Model, which represents the basis of the individual test cases. Depending on the type of derived test cases, this could save duplicated analysis effort and thus significantly improve performance in certain scenarios. We see possible applications for the presented concepts in the integration of the MCSTLC into a continuous test toolchain, where the ATE represents an essential building block of such a use case. Through this, a model-based development methodology can be extended with concepts of CI/CD to consistently ensure the quality of the generated model artifacts.

11

Model-Based Mutation Analysis

The last remaining step of the MCSTLC is the *Omni Model-based Mutation Analysis*, which is closely related to the *Test Case Generation* presented in chapter 9. Basically, in this process step, besides the guaranteed coverage of the Omni Model parts by the determined quantity of test cases, a statement about their quality shall be made. In this case, the statement is determined based on mutation of the considered system. In particular, new variants are created by changing parts of the original System Model, against which the considered set of test cases is subsequently executed. As already introduced in the corresponding foundations part (see section 5.1.3), for all mutants it is evaluated whether on the one hand there was a test case whose test result has changed noticeably compared to the reference run. On the other hand, for each test case, the amount of mutants killed is determined, which is further processed to a quality metric.

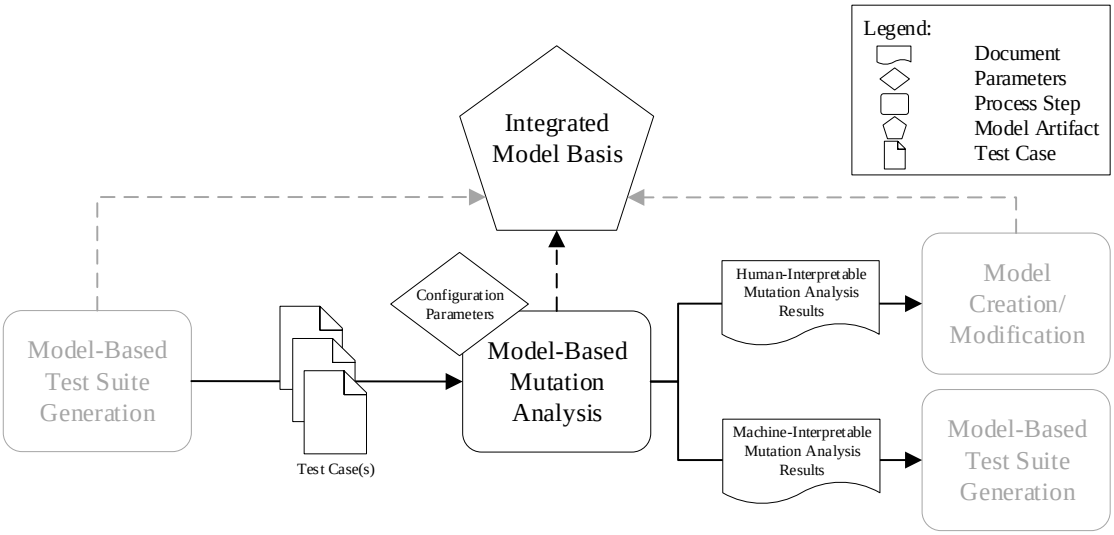


Figure 11.1: MCSTLC extract focusing the Mutation Analysis and involved information

Making use of the quality information about test cases, the process step is integrated into the MCSTLC as shown in figure 11.1. As shown on the left side of the figure, the previously generated test cases are part of the input for the mutation analysis. Besides, a set of configuration parameters is required. Concerning model information, the Integrated Model Basis is used again, which further provides a decisive advantage in

the sub-steps. After successful processing of the information, the results are prepared for two different usage scenarios which can be seen on the right side of the graph. These results provide aggregated information about the executions against the set of mutants.

11.1 Prerequisites for Mutation Analysis

In this section, a digression for basic mechanisms is conducted to keep the section self-contained. Furthermore, the available set of configuration parameters and input artifacts is discussed.

11.1.1 Digression: Mutation applied to the Execution Graph++

As already explained in section 5.1.3, a set of mutants is generated for the mutation analysis. These mutants represent modified variants of the original system and are derived from the original model using *Mutation Operators*. In particular, the set of Mutation Operators is usually tailored to a specific application domain. In the literature, many Mutation Operators, as well as languages for defining mutations, are presented [100]. Based on the results of literature research and the practical experiences from the T3 project (see section 2.2), a set of mutation operators is determined. Within the EGPP this set of mutation operators can be categorized according to their functionality. This is done in tabular form, whereby the respective functionality is explained in each case.

The first group of Mutation Operators, the *Node Mutations*, comprises structural mutations, which apply to the nodes of the EGPP structure. This type of mutation includes defects that represent incorrectly inserted, forgotten, or false statements of a code block.

The second group of *Transition Mutations* includes mutations affecting the structural nature of the EGPP model. In particular, defects are mapped that create a faulty control flow caused by incorrect positioning of code fragments.

The third group of mutation operators, the *Operator Mutations*, contains operators, which manipulate boolean conditions as well as mathematical expressions. Within this group, accidental mistakes of the developer are imitated, which have effects on the evaluation of the whole expression or the condition.

The fourth group of *Data Mutations* includes mutations, which replace concrete values and thus mimic an incorrect setting of constants or their datatypes.

The last group of mutation operators, the *SubGraph Mutations*, deals with possible faults in the context of the EGPP representation of partial functionalities such as method/-function calls. The stub concept, which was introduced in section 7.2.2, is explicitly

addressed here because potential faults can be introduced into the system by the user at this stage.

Altogether, the set of Mutation Operators shown provides a solid basis for the application of mutations in the EGPP context. On the one hand, the set of mutation operators presented in the literature is mapped on the model-level as far as possible. On the other hand, the specifics of the EGPP representation are considered so that extensive mutation possibilities can be realized.

Table 11.1: EGPP Mutation Operators

Category / Operator ID	Preconditions	Description	Pitfalls
Node Mutations			
`DELETE_NODE`	Target node is not a `EGPPInitialNode`, `EGPPFinalNode`, or a node with multiple but different numbers of incoming and outgoing transitions.	This operator removes the node from the graph and connects its predecessor with the successor.	
`DUPLICATE_NODE`	Target node is not a head- or tail-node of a loop.	This operator creates a copy of the targeted node and inserts it before or after the original node depending on the node type.	
`MAKE_FINALNODE`	Target node is not a `EGPPInitialNode`, `EGPPFinalNode`.	This operator converts the targeted node to a `EGPPFinalNode`.	
`CANCEL_FINALNODE`	Target node is not the last remaining node in the graph, or not a `EGPPFinalNode`.	This operator converts the given `EGPPFinalNode` into a standard `EGPPNode`.	
Transition Mutations			
`DELETE_TRANSITION`		This operator removes a transition from the graph.	The resulting graph may not represent a valid control flow.
`INSERT_TRANSITION`		This operator creates a new transition from a respective node to another random node of the graph.	The resulting graph may not represent a valid control flow.
`INVERT_TRANSITION`		This operator inverts the direction of the transition by swapping its start and end node.	
Operator Mutations			

Table 11.1: EGPP Mutation Operators (continued)

REDUCE_TO_ONE_PATH	Target node has more than one outgoing transition.	This operator removes all outgoing transitions of the node except for one random transition. It is a generalized version of REPLACE_COND_OPERATOR, but works on graphs with no code fragments at all.	The resulting graph may not represent a valid control flow.
REMOVE_ONE_PATH		This operator is similar to REDUCE_TO_ONE_PATH, but removes only one random transition instead of removing all except for one.	
EXCHANGE_COND_OPERATOR	Target node includes EGPPTaggedData with code fragments that contain at least one conditional operator. The following set of relational operators is supported: {>=, <=, >, <, ==, ! =} Further, these logical operators are supported: {||, &&, !}.	This operator exchanges one randomly chosen operator from the code with a second randomly chosen one while the types correspond.	
REPLACE_COND_OPERATOR		This operator is quite similar to EXCHANGE_COND_OPERATOR, but replaces a randomly chosen relational or logical operator and its children by a fixed Boolean value (true/false).	

Table 11.1: EGPP Mutation Operators (continued)

EXCHANGE_MATH_OPERATOR	Target node includes EGPP-TaggedData with code fragments that contain at least one mathematical operator. The following binary math operators are supported: {+, −, *, /, %} Additionally, these unary operators are handled: {++, −−}.	This operator exchanges one randomly chosen operator from the code with a second randomly chosen one while the types correspond.
REPLACE_MATH_OPERATOR		This operator is similar to EXCHANGE_MATH_OPERATOR, but replaces a randomly chosen operator and its children by a fixed numeric constant.
Data Mutations		
REPLACE_CONST	Target node includes EGPP-TaggedData with code fragments that contain at least one numeric constant.	This operator replaces one randomly chosen constant with a default value (e.g. '0').
MODIFY_CONST		This operator is similar to REPLACE_CONST, but modifies a random constant by e.g. adding or dividing a random number, instead of replacing it with a default value.

Table 11.1: EGPP Mutation Operators (continued)

MODIFY_VAR_DATATYPE	Target node includes EGPP-TaggedData with code fragments that contain at least one variable declaration. The supported datatypes are {*string*, *double*, *int*}.	This operator exchanges the datatype of a randomly chosen variable by another one.
Subgraph Mutations		
REPLACE_SUBGRAPH_CONST	Target node is of type EGPP-Graph, that represents a function/method call.	This operator replaces the subgraph by a standard node and the function call statement is replaced by a random constant.
REPLACE_SUBGRAPH_OTHER	Target node is of type EGPPGraph.	This operator replaces the subgraph by a random other subgraph node in the graph. The other subgraph is duplicated and the targeted is removed thereby.
REPLACE_SUBGRAPH_STUB	Target node is of type EGPP-Graph, having a corresponding stub graph.	This operator replaces the subgraph by its stub graph.
EXCHANGE_SUBGRAPH_OTHER		This operator is similar to REPLACE_SUBGRAPH_STUB, but exchanges the targeted subgraph with a random other one, instead of replacing it. The other subgraph is not duplicated and the targeted is not removed.

11.1.2 Configuration Parameters

As already mentioned at the beginning, Mutation Analysis offers a high potential for optimization in terms of both adaptation to the application context and performance. For this reason, some configuration options have been identified for the implementation of the Mutation Analysis at model-level, which are described in the following.

Mutation Operators Selection

The Mutation Operators contained in table 11.1 represent the complete range of available operators. In certain cases, like very early development phases, only a subset of the operators may be useful. For this reason, the expert can specify a subset of operators, which is then used for the Mutation Analysis. The subset is defined by a concrete selection of the operator IDs shown above. If the selection of the mutation operators is left to the algorithm, a suitable subset is determined internally based on characteristics of the graph structure (see section 10.1.1).

Mutant Sampling Approach

In the literature, many optimization options for Mutation Analysis have been presented, including the so-called Mutant Sampling. Driven by the goal of an applicable variant of Mutation Analysis at model-level, this technique is applied in the context of our solution. Furthermore, different variants of Mutant Sampling exist, of which a selection is implemented on the model-level. A distinction can be made between `RANDOM` and `EQUIVALENTS`, whereby the two variants are described in detail in section 11.2.2. If none of the mentioned approaches is desirable, the `OFF` parameter deactivates the functionality.

Mutation Strategy

Further, different variants for generating mutants by the combined application of Mutation Operators are presented in the literature. Some of these so-called Mutation Strategies have been implemented on the model-level and supplemented by an Omni Model-specific variant. The range is defined by `HIGHERORDER`, `HIGHERORDER_MIXED`, and `HIGHERORDER_MIXED`, which is discussed in detail in section 11.2.1. If none of the mentioned variants is desirable, the `CLASSIC` parameter represents the fallback to the standard approach of generating mutants.

11.1.3 Excerpt of the Omni Model

The correlations between the System Model and the Test Model stored in the Integration Model provide an advantage for Mutation Analysis. On the one hand the selection process of the mutation targets and on the other hand the selection process during the subsequent execution of the test cases against the created mutants. The minimum required Omni Model should therefore include the following models, System Model, Test Model and the Integration Model. Other domain-specific models currently do not provide any advantages for the Mutation Analysis. However, e.g linked error models like fault trees could be evaluated in future variants of a Model-Centric Mutation Analysis to make the application of mutations more targeted.

11.2 Mutation Analysis

The Mutation Analysis is divided into three main disciplines, namely *Mutant Generation*, *Mutant Execution*, and the final *Mutant Execution Result Evaluation*.

In the first step, the reference test results are initially determined for the unmodified system. Each test case of the test suite is evaluated against the System Model using the Data Flow Aware Abstract Test Execution (DFA-ATE) approach presented in chapter 10. Based on sufficient test verdicts, namely `PASSED` or `PROBABLY PASSED`, the set of test cases is determined which are subject to the Mutation Analysis. At this point, the actual generation of mutants starts, utilizing the configuration parameters defined. The relevant part of the System Model is identified from the test cases via mappings in the Integration Model. The focused part of the System Model is analyzed according to the selected Mutation Strategy and a set of mutation targets is derived. The identified mutation targets are compared with the specified set of mutation operators, whereby operators not applicable are omitted. Finally, the mutants are created by applying the operators to the identified mutation targets according to the chosen Mutation Strategy.

In the second step, the *Mutant Execution*, further optimization measures are implemented. The sampling variant specified by the expert is applied, thus reducing the number of mutants to be analyzed. Clustering is performed, which in turn identifies a set of test cases based on information from the Integration Model, whose test results could potentially be changed by the mutant. Once this clustering is complete, the identified test case sets per mutant are executed using the approach presented in chapter 10 to perform an Abstract Test Execution (ATE).

In the last step, the insights collected in the ATE are evaluated and transformed into the respective results, which are further processed by the other process steps of the MCSTLC. In particular, the Mutation Score is calculated for the test cases, whereby the Equivalent Mutants problem is explicitly addressed. Furthermore, the collected information of each ATE run is prepared and enriched with metrics, which provide

the user/expert with helpful information for further improvement of the current test suite.

11.2.1 Mutant Generation

According to the subtasks described in the overall description, the mutant generation is divided into the following parts.

Reference Execution and Integration Model-Based System Model Selection

Starting from a test case set created in the Model-Centric Test Case Generation, reference results are determined by executing each test case against the original System Model. This System Model is taken from the Integrated Model Basis and can be uniquely determined by the specified mapping between Test Model and System Model (via Integration Model). After successful execution of ATE, the generated test reports are evaluated. In particular, the included test verdicts are used as selection criteria to determine which test cases are used for the Mutation Analysis. In literature, usually, only the test cases with a `PASSED` verdict are used. Due to the more abstract context and the uncertain information situation in the early phases of development, another Test Verdict (`PROBABLY PASSED`) was introduced (see section 10.2.5). Test cases assigned this Test Verdict in ATE could not be evaluated to `PASSED` with absolute certainty. This is due to included data that could not be assessed until now. However, there is significant evidence, that the outcome is `PASSED`, in case of all the data may be assessed. For the use case of Mutation Analysis, such classified test cases are just as valuable as the `PASSED` test cases, since in the worst case a mutation affects an aspect that cannot be evaluated, which would however lead to an Equivalent Mutant afterward. Thus, such test cases do not affect the Mutation Analysis negatively. Test cases that have received one of the remaining Test Verdicts (`FAIL`, `INCONCLUSIVE`) based on the reference execution are not used for the Mutation Analysis. However, the reports for such test cases are made available to the user, whereby improvements for the System Model, as well as the Test Model, can be derived.

After successful evaluation of the results of the reference executions, the set of test cases that are processed by the Mutation Analysis is determined. Since the goal of the Mutation Analysis is not to find new test cases through the mutations, but to evaluate the quality of a given set of test cases, the targeted nature of the mutants plays an important role. In this case, *targeted* means that mutants are designed in such a way that the mutated model elements are triggered by the respective test case if possible. Therefore, the extensive model information of the Integrated Model Basis is used to identify corresponding parts of the System Model. Figure 11.2 shows an example of the relationships between the individual model artifacts and how they are used.

This method of identifying the appropriate System Model was already used earlier in the reference executions. In the first step, it is determined for a test case from which

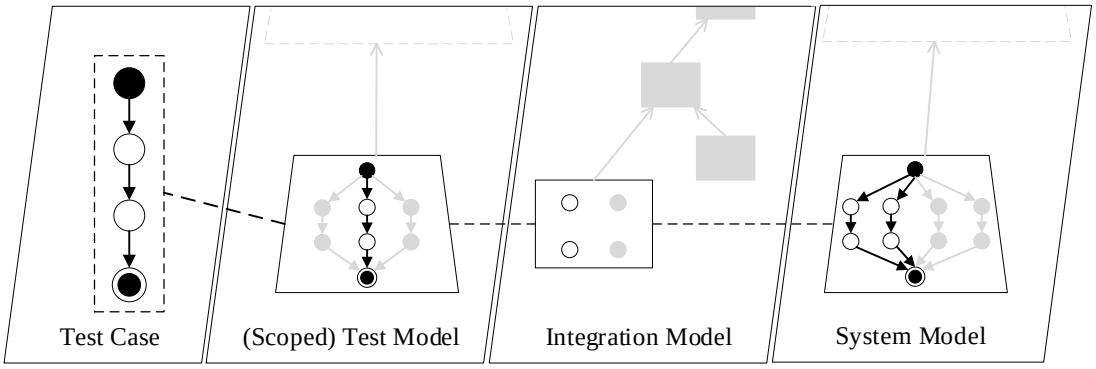


Figure 11.2: Mapping of model artifacts across domains of the Integrated Model Basis

Scoped Test Model it was generated, i.e. from which submodel of the comprehensive Test Model it emerged (see the left side of figure). In the next step, the links between model artifacts of the Test Model and the System Model stored in the Integration Model are used to extract the transitively affected System Model parts (right side of figure). In the last step, an aggregation of all extracted System Model parts is performed, whereby a valid Sub-System Model is determined in the case of higher integration levels for a possibly fragmented Subsystem Model.

Mutation Strategy-Based Mutation Targets Determination

After the relevant parts of the System Model have been identified for the respective test case, the previously selected Mutation Strategy is applied. A Mutation Strategy tries to reduce the number of the provided mutants and thus necessary test case executions. By the reduced number of mutants against which the test cases are executed, we expect the expressiveness remains constant, shown for the code-level [142]. In addition to the classical application of mutation operators to generate mutants (`CLASSIC`), three strategies are implemented for the model-level. Each of these strategies applies a set of mutation operators according to various rules. The rules include the selection of possible targets for the application of the operators.

The first strategy (`HIGHERORDER`) creates a mutant by applying the same Mutation Operator multiple times to different Mutation Targets of the System Model. In contrast to the `CLASSIC` strategy, which applies a Mutation Operator to only one Mutation Target to produce a new mutant, theoretically, any number of applications of an operator can result in a new mutant. To control the number of applications per Mutation Operator, our concept includes the specification of a percentage value that indicates how high the proportion of actually mutated components is compared to the number of potential mutation targets. Besides this variant for the numerical restriction of mutations per

mutant, there are no options for restriction in this strategy.

The second mutation strategy (`HIGHERORDER_MIXED`) is designed similarly, but different mutation operators can be included. I.e., the mutants represent the result of a combined application of different mutation operators. Analogous to the `HIGHERORDER` strategy, it is possible to cap the number of mutation operators leading to a mutant by a percentage. How mutation operators can be applied to the System Model to produce a mutant is only subject to the constraints mentioned in table 11.1.

In contrast, the mutation strategy `HIGHERORDER_CS` introduces a new way of constraining mutation targets. Different mutation operators can be applied in combination. However, it is a *context-sensitive (CS)* strategy that allows mutation targets that are not part of the same control flow to be mutated in combination. This type of Mutation Strategy is motivated by the fact that emergent effects between mutations are minimized by such mutants. The proposition of such a killed mutant is therefore more promising in terms of identifying modeling problems and resulting improvements. The Mutation Strategy, therefore, yields, as a result, several sets of model elements, where the elements of one set do not affect each other since they are part of independent control flows of the System Model by definition. Figure 11.3 illustrates this additional constraint on the choice of Mutation Targets on the System Model excerpt *CSM_ON* of the Running Example, which is revisited here and reduced to the necessary information, the mere control flow.

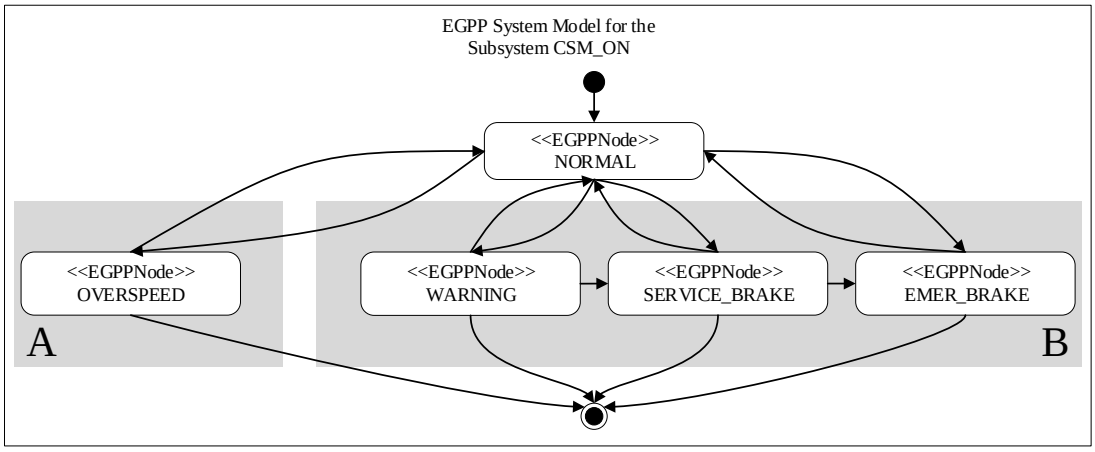


Figure 11.3: Illustration of sets of independent model elements for the `HIGHERORDER_CS` Mutation Strategy (reduced model)

Analyzing this System Model concerning Mutation Targets for the `HIGHERORDER_CS` strategy, first, the paths between an `EGPPInitialNode` and an `EGPPFinalNode` are determined. All paths with multiple iterations through a contained loop are excluded. However, a single iteration through a loop is allowed. From the resulting set of paths, the elements that are part of multiple paths are each added to a separate set (see `«EGPPNode» NORMAL`). For such model elements, no combination with other elements can be made, since there is always the possibility of emergent effects during the ATE.

Based on the assumptions explained above, the remaining EGPPNodes and connected EGPPTransitions can be divided into two groups, each of which is highlighted in gray in the graph (see A and B). If one takes a model element from set A and a model element from set B, the two mutation targets are guaranteed to be free of mutual influences and can therefore be mutated in combination. The possible sets of mutation targets that can be combined to form a mutant represent arbitrary pairings of two model elements of the addressed sets. It is not necessary to consider all permutations between the sets because no additional information can be derived, but the number of possible mutants would be increased enormously. This contradicts the actual goal of a reduction of the mutant set compared to the CLASSIC strategy.

Altogether, the Mutation Target Determination identifies several sets of Mutation Targets, where the mutation of all model elements contained in a set produces a mutant. In short, the Mutation Strategy CLASSIC consists of sets containing a single model element. In contrast, the Mutation Strategies HIGHERORDER and HIGHERORDER_MIXED each create a set that contains all possible Mutation Targets. The last Mutation Strategy (HIGHERORDER_CS) represents a combination of the other strategies, with the resulting sets containing only Mutation Targets that do not affect each other.

Expert-, Experience- and Model Characteristics-Based Mutation Operator Selection

Based on the sets of Mutation Targets determined in the previous step, the corresponding Mutation Operators are determined in this step. The selection of operators used to create mutants depends on the following factors.

Primarily, the choice is reduced by the characteristics of the respective System Model. In particular, the categorization of EGPP models introduced in section 10.1.1 is used. If the System Model is categorized as Level 0 (Fragmented Control Flow Graph) or Level 1 (Control Flow Complete Graph), only Mutation Operators targeting structural/control flow components of the EGPP model are practicable. Specifically, due to such categorization, operators from the following sets are available: Node Mutations, Transition Mutations. If the System Model is categorized as Level 2 (Data Flow Graph), all remaining sets of Mutation Operators (Operator Mutations, Data Mutations, Subgraph Mutations) are supported. This excludes Mutation Operators that are not applicable due to missing features of the model. Besides, during the subsequent application, i.e. the actual generation of the mutants, some of the Mutation Operators are omitted, since the constructs in question are not expressed in the model.

Moreover, experts can make a selection of the shown Mutation Operators. This allows the expert to make explicit use of the optimization concepts *Selective Mutation/Constrained Mutation* and thus improve the entire Mutation Analysis. These concepts reduce the set of Mutation Operators and thus the set of resulting mutants [133][175]. In addition to implementing Selective Mutation, this mechanism enables the expert to select use case-specific operators. This ensures the best possible imitation of typical error cases in the context of the use case.

The last aspect of Mutation Operator Selection is the algorithmic preselection of operators. In particular, this kind of selection takes place when the expert has not specified any concrete subset. For the different characteristics of a System Model (`Level 0-2`) predefined subsets of the total set of operators are available. These represent in each case a set of operators that is as broad as possible, but as minimal as possible in terms of number. The predefined subsets are created based on expert knowledge. For example, the predefined set for the `Level 1` is composed of the following operators: `DELETE_NODE`, `DUPLICATE_NODE`, `DELETE_TRANSITION`, `INSERT_TRANSITION`

Taking the cut set of the three explained factors to reduce the Mutation Operator set, we obtain the set of operators that is finally used for the generation of mutants. To make the concepts more tangible, the System Model from figure 11.3 is taken up and transformed into a set of mutants. Based on the included model elements, this System Model is classified as `Level 2`, which initially does not restrict the choice of Mutation Operators. However, in this case, the expert has decided to use the `DELETE_TRANSITION` operator to generate mutants. Furthermore, the `HIGHERORDER_CS` strategy was chosen, generating the set of mutants indicated in figure 11.4.

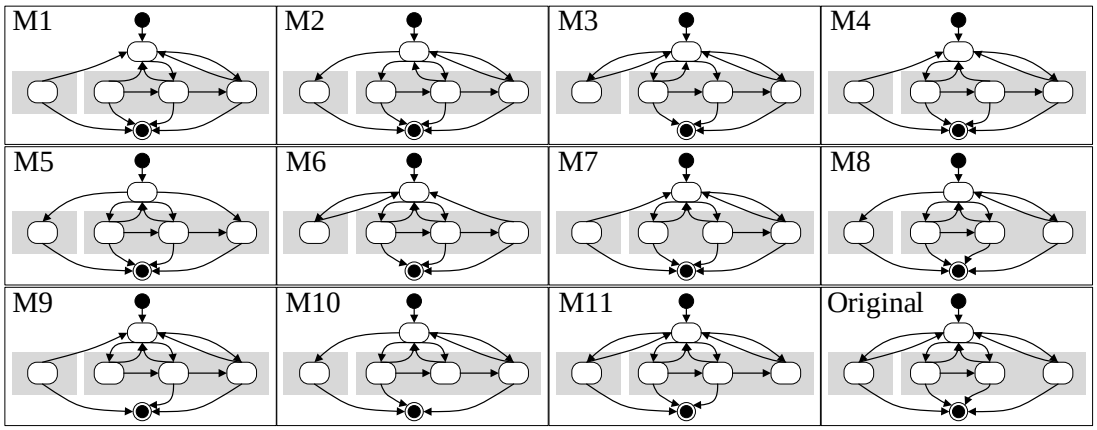


Figure 11.4: Generated Mutants for Running Example as per `HIGHERORDER_CS`

In total, the application of the Mutation Strategy `HIGHERORDER_CS` with the Mutation Operator `DELETE_TRANSITION` results in 11 mutants. As described above, one mutation of a model element from `A` and one mutation of a model element from `B` are each applied in combination to produce a new mutant. In comparison, the `CLASSIC` strategy with the same Mutation Operator applied would have 14 mutants, since each mutation produces a new mutant.

11.2.2 Mutant Execution

After the successful creation of mutants, the mutants are executed against the determined test case set. Each test case could be executed against the determined mutants using ATE. The selection of a representative subset of the mutants, the selective execu-

tion of test cases against mutants, as well as the actual execution itself is explained to optimize the approach.

Mutant Sampling

Depending on the selected Mutation Strategy, a large or very large set of mutants is the starting point for this processing step. Optimizations should reduce this quantity without losing significance. For the application context of code-based Mutation Analysis, several approaches have been presented [123][176]. Based on the ideas, two approaches are developed for the model-level that accomplish a mutant set reduction in different ways.

The first possibility to implement sampling on the mutant set is a randomized selection procedure (`RANDOM`). Here, randomly selected mutants are removed from the set until a reduction is achieved such that the remaining set is only a pre-specified percentage of the original set. At the code-level, a large number of experiments have determined that, for example, reducing the mutants to 10% of the original set, produces only a 16% degradation in effectiveness [100]. For our concept at the model-level, the same general conditions apply to a large extent and therefore we expect analogous key indicators.

The second option for reducing the mutant set primarily utilizes the atomic mutations and properties of the mutants and is developed against the background of effective interaction with the subsequent Mutant Clustering. The focus of this variant is on the elimination of similar mutants in terms of the combination of mutation operator and mutation target (`EQUIVALENTS`). All information is captured during the creation of the mutants and can therefore be used for the selection process. To make the selection process stable against errors, the set of mutants has to be sorted in descending order by the number of applied mutations beforehand. Afterward, the list is examined and mutants that are included in other mutants are removed from the list. *Included* means that in an already examined mutant all pairs of mutation operator and target are applied, which the currently examined mutant also contains. Thus, these mutations are already covered by the more comprehensive mutant, so no new findings are expected from this mutant. Once the list of mutants has been fully examined, the reduced set of mutants determined by the `EQUIVALENTS` sampling is determined.

If the all-encompassing set of generated mutants is to be used for the Mutation Analysis, sampling can be omitted. However, this can result in a significantly higher execution time.

Mutant Clustering

Another optimization is Mutant Clustering. Analog to the field of Mutant Sampling, there are code-based Mutant Clustering methods, which can be shifted to the model-level [120][54]. In the code-based clustering approaches, clusters are formed by group-

ing mutants that are guaranteed to affect the same set of test cases. From these clusters usually, only some representatives are chosen to be used for the subsequent Mutation Analysis. This approach, initially proposed by Hussain et al. [97] has been refined in many variants and serves as a starting point for our concept.

The Mutant Sampling variant `EQUIVALENT` and our Mutant Clustering approach are closely related. The `EQUIVALENT` sampling variant covers the second aspect of code-based clustering methods, namely the selection of suitable representatives. Specifically, in our case, those mutants represent the superset of other less comprehensive mutants. The actual task of the model-centric Mutation Clustering concept is therefore to implement the first aspect of the code-based clustering procedures, namely the grouping of test cases for suitable mutants. The focus is on the information stored in the Omni Model, which has already been used for the targeted identification of mutation targets. Figure 11.5 illustrates the essential aspects of grouping.

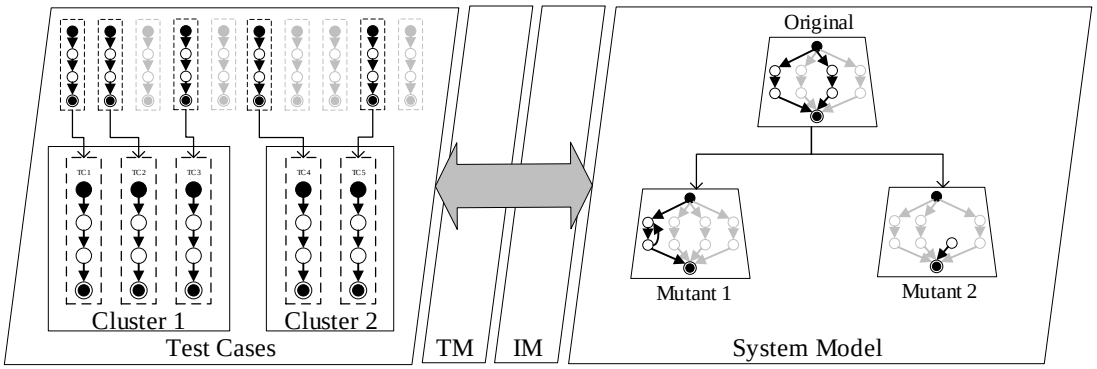


Figure 11.5: Schematic figure for the identification of test cases clusters per mutant

On the right side of the figure, the System Modeling area can be seen, showing, on the one hand, the original System Model, and on the other hand, the mutants created in the previous step. The middle area of the figure shows both the Integration Model (IM) and the Test Model (TM), which are part of the Omni Model and form the bridge between mutants and test cases in this application context. On the left side, the test cases are shown. In the upper pane, the set of test cases created for the original System Model shown on the right are illustrated. In the lower pane, one can see the two created clusters `cluster 1` and `cluster 2` corresponding to mutants `mutant 1` and `mutant 2` respectively. A test case is inserted into `cluster 1` exactly when model elements of the mutant control flow path from `mutant 1` are connected to model elements of the respective test case via the Integration Model. This way, it is very likely that the test case triggers the mutated part of the System Model in the later execution and thus can prove to kill the mutant, i.e., a change is detectable in the test report. The same applies to all other mutants and the resulting clusters. Depending on the Mutation Strategy applied before, it is possible that a test case is part of several clusters, which means that the intersection of the clusters does not have to be empty.

Overall, this type of Mutant Clustering in combination with the previously introduced Mutant Sampling ensures that the Mutation Analysis can be performed with maximum

efficiency. In particular, only the test cases relevant for the mutant are included in the cluster, executed, and finally evaluated to a Mutation Score (see section 11.2.3).

Parallel Abstract Test Execution

Finally, we discuss the use of ATE in the context of Mutation Analysis. The test cases are evaluated against the mutants by the interpreter-based approach presented in chapter 10. This approach offers the major advantage that a separate executable version of the system does not have to be compiled for each mutant created, but the mutant model information is processed directly. Thus, the execution is less performant than comparable compiled code-based executions but can be applied in early development stages.

The execution of the test cases can be performed against the mutants utilizing the mechanism shown in chapter 10. Based on the previously determined clusters per mutant, parallelization of the execution can be applied. Therefore increasing the scalability, the individual executions must be independent, which is the case by definition.

In addition to parallelization, other approaches for improving performance are discussed in code-based Mutation Analysis approaches. For example, in contrast to the original execution concept of *Strong Mutation*, which makes the result of Mutation Analysis dependent on a changed system state at the end of the execution, the variants *Weak* and *Firm Mutation* were presented [95][177]. These variants check the internal system state throughout execution and abort execution if a mismatch is detected, causing the mutant to be considered as *killed*. In our concept, such variants are not applied, since they would negatively affect our approach to automated detection of *Equivalent Mutants* in the course of Mutation Score computation (see section 11.2.3).

11.2.3 Execution Result Evaluation

After all executions of test cases against mutants by the ATE approach are completed, the collected findings are evaluated. This includes the context-sensitive computation of metrics and the appropriate preparation of detailed information for the respective follow-up purposes.

(Test Case) Mutation Adequacy Score Computation

After successful execution of all test cases against the respective mutants, a metric is calculated describing the adequacy of the test suite. This so-called Mutation Adequacy Score (MAS) indicates the killed mutants with the total set of mutants for a test case set. In particular, the calculation of the MAS takes into account the so-called *Equivalent Mutants* by subtracting them from the total set of mutants. However, the detection

of these Equivalent Mutants is a core challenge of Mutation Analysis, which can be identified by experts [100].

As part of our approach to Model-Centric Mutation Analysis, the detection of Equivalent Mutants can be automated, based on the execution logs generated by the ATE approach. This enables the expert to automate the identification of Equivalent Mutants during the execution. As shown in chapter 10, such an execution log contains the sequence of model elements passed during the traversal of the System Model and the sequence of model elements of the respective test case including detailed information about processing steps of the ATE component. Further, for all model elements, the included meta information, such as the evaluated code fragments, is recorded. If two executions of a test case lead to the same Test Verdict, it is examined whether an *equivalent mutant* is present. Therefore, the logs of the executions are compared. If the logs just differ in the code fragments or model elements that were changed by the mutation operators, it is an Equivalent Mutant. If other differences are found, this indicates a change in control or data flow caused by the mutant and thus an altered internal state of the system. Therefore, it can be concluded that the test case is not capable of detecting the mutation. This can either be since the test case does not check this type of fault or is of insufficient quality, i.e. does not cover the mutated parts.

To increase the amount of information provided by a successful Mutation Analysis and to obtain a statement on the adequacy of individual test cases, a supplementary metric is developed for the model-level, the so-called Test Case Mutation Adequacy Score (TCMAS). For this purpose, the perspective of the particular test case is taken to determine the mutants it was executed against. Based on this subset of mutants, the ratio of the killed mutants to the total subset is determined. Equivalent Mutants are determined according to the already presented procedure and considered in the calculation formula accordingly.

Definition 36 (**Test Case Mutation (Adequacy) Score**)
This metric is defined as follows:

$$TCMAS = \frac{D}{M - E}$$

with

- D*: number of mutants killed*
- M*: number of mutants the test case was executed against*
- E*: number of equivalent mutants the test case was executed against*

This metric makes the contribution of each test case to the overall MAS transparent, used as feedback for the creation of test suites based on it (see chapter 9). It is expected to significantly increase efficiency, in terms of smaller and more effective test suites, especially in advanced iterations of MCSTLC. A combination of mutation metrics with coverage metrics enables a simplified identification of weak points in the test suite. The expert can then make appropriate adjustments to the affected model parts with the help of the Omni Model information.

In addition to the presented mutation metrics, further metrics are conceivable to improve the informative value. A possible extension could be a metric describing the diversity and heterogeneity in the applied mutation operators of a mutant. From this, it could be derived to a certain extent how difficult it is to kill the respective mutant and how high quality a test case is that has killed the mutant.

Human- and Machine-Interpretable Advice Generation

As shown at the beginning of this chapter, the results of the Mutation Analysis are made available to the *Test Case Generation* and the *Model Creation/Modification*. The pure machine and automated processing of the information within the *Test Case Generation* do not require any special preparation of the collected information, but only well-defined data structures to exchange the information. However, to facilitate the processing of the information by the modeling expert in the context of *Model Creation/Modification*, a *Mutation Analysis Report* is generated from the collected data, which consists of:

- Overall report on the test suite under investigation
  - Composition of the test suite
  - Detailed information about the mutants
  - Achieved MAS of the test suite
- Detail report per test case
  - Achieved TCMAS of the test case
  - Execution report per mutant
    - Performed substeps of the test case
    - Traversed path in the System Model/mutant
    - Detailed execution log of ATE component
    - Test verdict determined by the ATE component

Metrics like the TCMAS can be found in this list. The individual aspects of the report are presented in a textual form and, in conjunction with the model elements from the Integrated Model Basis, provides valuable information for model-side improvements. Corresponding components of the ATE report have been shown in section 10.2.6. Furthermore, the relationships between the modeling domains specified in the Integration Model can be used to uncover modeling weaknesses outside the Test and System Model and to correct them at an early stage. Such possibilities represent another special feature of the Omni Model-based Mutation Analysis approach and cannot be implemented easily in classical contexts.

11.3 Technical Realization within A3F

For the MCSTLC process step just presented, there is a prototypical implementation of the concepts in the context of the A3F. The analysis, which encapsulates the functionality, in turn, provides the user with several parameters that are used to configure the internal processing chain and supply it with data. On the one hand, the analysis results of upstream processing steps are used via the parameters. On the other hand, the expert can specify parameters to adapt the processing chain to the conditions or to optimize it for the respective model instances.

Table 11.2 gives an overview of the available parameters, explains them briefly and shows the underlying concepts.

Table 11.2: Configuration parameters for `egpp_mutation_testing` analysis

Parameter	Description
`integrationmodel`	Determines the Integration Model which manages the Omni Model around the `systemmodel` and the `testmodels`
`systemmodel`	The System Model which represents the basis for the application of mutations
`testmodels`	The Scoped Test Models from which the `test cases` are derived
`testcases`	Determines the set of test cases which are processed by the mutation analysis
`executor`	Determines the executor, for the Abstract Test Execution. Possible values are `controlflow` and `dataflow`
`executorconfig`	Determines a set of configuration parameters for the respective `executor`
`seed`	A random seed to make all of the processing reproducible
`mutopconfig`	Determines the set of applied mutation operators. If the `AUTO` option is specified, a predefined set of operators is configured. Otherwise, the set of mutation operators is determined by the passed set of operator IDs (see section 11.1)
`mutsamplingconfig`	Configures the sampling task of the mutation analysis. `OFF` deactivates sampling, while `RANDOM` and `EQUIVALENTS` each together with a double value between `0` and `1` pick certain samples (see section 11.2.2)
`mutstrategyconfig`	Configures the applied mutation strategy during mutation analysis. Possible values are `HIGHERORDER`, `HIGHERORDER_MIXED`, `HIGHERORDER_CS` and `CLASSIC`, while the first two each can specify a double value between `0` and `1` (see section 11.2.1)

Note the overlap with the `egpp_execute` analysis presented in the last chapter. The parameters `integrationmodel`, `systemmodel`, `testmodels`, `testcases`, `executor` and

executorconfig are identical to those presented in section 10.3. This is especially since internally, in the context of mutation analysis, the same execution mechanics are used.

The remaining parameters seed, mutopconfig, mutsamplingconfig, and mutstrategyconfig therefore specifically concern the mutation mechanism and the associated processing logic. The concrete expressions of the parameters can be taken from the table shown above or found in the respective concept part.

As mentioned above, successful mutation analysis in the context of A3F requires the combination of several atomic analyses. Again, parts of the already created combinations of analyses are reused, as shown by the grayed-out parts in figure 11.6.

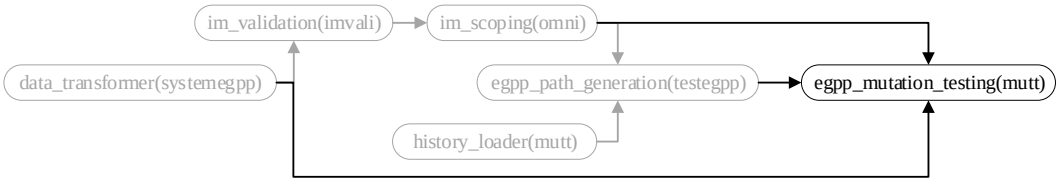


Figure 11.6: Analyses dependency graph for the egpp_mutation_testing analysis

The graph shows which analyses represent the input values for further processing within the egpp_mutation_testing analysis. Listing 11.1 shows a specific configuration of the egpp_mutation_testing analysis. The parameters presented in the previous section are assigned concrete values, such as those that may occur in practical use in the course of A3F.

```
<configurations of preceding analyses>

analysis egpp_mutation_testing(mutt) {
  integrationmodel=
    "im_scoping|omni|IMScopingResult|getFilteredIMModel";
  systemmodel=
    "data_transformer|systemegpp|DataTransformationResult|getOutputElements";
  testmodels=
    "im_scoping|omni|IMScopingResult|getExtractedMBTModel";
  testcases=
    "egpp_path_generation|testegpp|EGPPPathGenerationResult|getTestCases";
  executor="dataflow";
  executorconfig="10,10";
  seed="111";
  mutopconfig="AUTO";
  mutsamplingconfig="RANDOM,0.2";
  mutstrategyconfig="CLASSIC";
}
```

Listing 11.1: Example configuration for the egpp_mutation_testing analysis

After the execution of such a configuration by the A3F, the framework generates numerous result artifacts. This includes reports about the execution of the test cases against the different variants of the original System Model. Furthermore, a comprehensive report is generated that shows metrics and reveals the relationships. For the

Running Example, which has already been used to illustrate the concepts across all other chapters, excerpts of concrete result artifacts of the prototypical implementation have already been shown in the concept sections (see listing 10.1). In section 13.4 a more comprehensive example of a Mutation Analysis report can be seen.

11.4 Related Work

In addition to our concept on model-centric Mutation Analysis, there are a large number of research contributions in this context. In general, there is a very positive trend in publications in this area over the last years [100] [135]. This underlines the interest in the subject area whereby a steady improvement of certain aspects of the basic concept is achieved and solutions for practical use emerge. This research area covers a large number of research topics due to the variety of activities included, while only a selection of aspects is discussed in the context of this Related Work. Since the majority of existing research contributions are based on specific programming languages, a selection of these frameworks and their use cases is considered first. Due to our modeling focus, alternative approaches that enable mutation analysis at the model-level or through domain-specific languages are discussed. Finally, we consider further optimization techniques in the context of Mutation Analysis, which are not applied in our context but bring an enormous improvement in other use cases.

Mutation Analysis Frameworks on the Code-Level and its Industrial Applications

As mentioned above, there is a Mutation Analysis framework or at least concepts for the implementation of such a framework for almost every existing programming language. For example, Delgado et al. have studied several mutation operators using the framework `MuCPP` in the context of the C++ language [52]. Besides, there are some very specific approaches, such as the so-called `CCmutator`, which determines mutants specifically for parallel execution strings in C/C++ [116]. The popular Java programming language has many frameworks that implement Mutation Analysis. Common representatives are `muJava` and `Pit` [122][45]. In addition to the common mutation operators that are uniform across most programming languages, additional mutations are usually defined, especially in object-oriented languages. For this purpose, Offutt et al. have defined a set of so-called *Inter-Class Mutation Operators*, which aim at the mutation of concepts such as polymorphism [121]. Overall, a very large variety of mutation operators is available, whereas our EGPP-based implementation of mutations relies on atomic faults in the context of the control- and data flow of model elements and our integrated code fragments. This enables our approach to mimic complex operators in a very specific context but keeps the applicability in heterogeneous contexts.

In practical development scenarios, the challenges of Mutation Analysis, especially the cost of execution as well as the selection of good mutants, are repeatedly highlighted. It was shown, for example, by Petrovic et al. in an evaluation of several applications

of Mutation Testing in an industrial setting [138]. In particular, this put the claim of a Mutation Adequate Test Suite into perspective and addressed the number of *unproductive mutants* in such test suites. For example, Ahmed et al. applied Mutation Testing to a test suite of a Linux kernel module [16]. They were able to identify gaps in the test suite and even found previously undiscovered bugs in the same process.

Mutation Testing on the Model-Level

In addition to the approaches to Mutation Analysis on specific programming languages, there are some approaches for modeling languages or domain-specific languages and their metamodels. The first tool to be mentioned here is MoMut::UML, which emerged from a research project lasting several years and enables Mutation Analysis of UML models of the SUT [115]. Due to the challenges of Mutation Analysis at the model-level, optimizations regarding mutant selection are implemented by this tool. Concepts for Mutation Testing or Mutation Analysis have been implemented for other popular modeling languages. In particular, some approaches to statemachines or statecharts, but also System Modeling using Petri nets are available [63][91][62].

Against the background of such Mutation Analysis concepts for modeling languages, the foundation for our EGPP-based Mutation Analysis was laid. In particular, our concept focuses on applicability across a variety of modeling languages. In this context, the domain-specific language *Wodel* should be mentioned, which allows defining mutations on metamodels and thus provides a flexible mechanism and the basis for applying Mutation Analysis to almost arbitrary models [76]. Building on this language, a framework was created by Gomez-Abajo et al. that provides the necessary functionalities to implement Mutation Analysis using the Wodel framework [77]. Compared to our concept, the focus here is on the adaptation of the mutations to the target model, whereas in our approach the target model is mapped to a uniform internal representation utilizing transformations, leaving the mutation definitions unchanged. With this concept, we can implement a well-tested core framework for mutation and further adapt new application contexts by simply specifying new model transformations.

Optimization Techniques for Mutation Testing/Analysis

Regardless of the specific modeling or programming language that represents the basis for Mutation Analysis, the focus is on the efficiency of the respective approach. In the previous sections, some approaches for optimizing Mutation Analysis have already been presented in the context of our concept. However, the range of research contributions on further approaches is very extensive, which is why further insight is given here. Pizzoleto et al. have evaluated a large number of research items that deal with optimization in the context of Mutation Analysis [140]. In particular, they created categories that describe and group the different optimization techniques.

For example, the use of Evolutionary Algorithms is mentioned as an alternative to the

mutant set reduction or even test case set reduction techniques we utilize. Abuljadayel et al. use such an algorithm to create promising Higher Order Mutants, where the fitness function describes how many test cases in a set were able to kill a mutant [14]. The mutation operators used to create the mutants represent another prominent area of research. In particular, the focus is on the selection of a targetable subset of the operators, as demonstrated by the concept of *Sufficient Operators*, which has been studied by Just et al. [106], among others. This concept is closely related to the variants of *Constrained Mutation* and *Selective Mutation* mentioned earlier. However, the range of concepts is almost unlimited at this point, which makes a holistic representation impossible. Already during the creation of the mutants, in most cases, potential Equivalent Mutants are examined. For example, Kintis et al. used compiler detection mechanisms to identify them before they are executed [111]. In contrast, in our approach, checking for Equivalent Mutants is only possible after successful execution by our ATE. Against the background of faster execution of code-based mutants, the concept of *Metamutants* is very popular. A *Metamutant* represents a parameterizable variant of the original system that includes all mutants, whereby configuring the parameters, the concrete mutant is expressed [78]. Such approaches reduce the overhead of executing the mutants, providing a significant overall advantage.

Overall, all approaches to optimize Mutation Analysis can be divided into the categories *Mutant Reduction*, *Equivalent Mutant Detection*, *Faster Execution*, and *Reduction of Executions*. From each of these categories, a concept applicable at the model-level is implemented to improve our Model-Centric Mutation Analysis in terms of its applicability. In general, however, code-based approaches and corresponding optimization concepts are much more prevalent compared to the model-level approaches.

11.5 Conclusions and Outlook

In this chapter, we present an approach that implements the concept of Mutation Analysis based on the Omni Model. The focus is on the evaluation of the test cases created in the process step *Test Case Generation* concerning their quality of detecting artificially inserted faults. In response to the question formulated at the beginning

How could mutation testing concepts together with a mechanism for test case execution applied on the model-level improve the quality of automatically generated test suites?

the following answer and concluding statement can therefore be given.

To make the concept of Mutation Analysis applicable to our internal metamodel for representing the System Model, a set of mutation operators is created and implemented. This enabled the creation of mutants, which is an essential aspect of this concept. Together with the concept presented in chapter 10 for executing test cases against a System Model, the foundation for model-centric Mutation Analysis is laid. Based on a variety of findings on the computational integrity and scalability of the approach, which are

available in the literature on code-based approaches, a concept for optimizing our concept at the model-level is created. Essentially, established mutant set reduction techniques were adapted to the model-level and improved against the background of the comprehensive Integrated Model Basis data. Besides, a further enhancement of scalability is implicitly introduced by the ATE technique, further improving the applicability of the approach. The metrics for the quality of a test case set, which are typical for Mutation Analysis, have been completed by newly designed metrics, whereby overall a clear picture of the quality of the test case set at hand can be created. In interaction with the modeling expert, as well as in the automated context with the *Test Case Generation*, a continuous improvement of the model basis, as well as the resulting test suite can be achieved. Also, Mutation Analysis is a useful addition to the established coverage metrics, allowing a more comprehensive quality and completeness statement to be generated.

In addition to the positive aspects of Model-Centric Mutation Analysis mentioned above, there are some challenges. First of all, the Integrated Model Basis has to be mentioned again. Here, analogous to the findings of the previous chapters, the quality of the model information has a not inconsiderable influence on the applicability and resulting outcomes of Mutation Analysis. As a concrete example, the described mechanisms for the goal-oriented selection of mutation targets and the implementation of the Mutation Strategy `HIGHERORDER_CS` are to be mentioned at this point. Another challenge is given in the specification of the input parameters by the modeling expert. In this case, it is always possible to fall back on the non-optimized variant or automatic default parameters, but this is at the expense of quality and/or performance in the specific application. In the case of a manual specification of these parameters, however, experience and knowledge of the application context are necessary, since the choice of parameters otherwise paints a false picture of the quality of the test case set. The last thing to mention is the lack of possibility of a purely manual application of mutations to the System Model, which is not provided for in our concept. Thus, the modeling expert cannot complete an automatically created set of mutants with mutants that seem reasonable by experience. However, such functionality can easily be added in future variants, giving the approach even more flexibility.

Provided that the aspects mentioned do not pose a problem in the specific use case, we see a future scenario of our model-centric Mutation Analysis in the embedding into a Continuous Integration process on the model-level. As the overview for the MCSTLC already suggests, continuous execution of this approach in parallel to other development processes is a good idea and provides valuable information about the quality of the current test suite. Further, we see the potential for machine learning-based approaches to tackle the selection of a proper mutation operator set for the application scenario at hand. In general, machine learning is a meaningful technology to boost disciplines of Mutation Analysis, as long as there is enough data for the concrete problem domains.

Part IV

APPLICATIONS AND EVALUATION

12

Applications of the Omni Model Approach

Following the main part of the thesis, the concepts that have been developed are evaluated in the following chapters. The qualitative evaluation of the Omni Model approach marks the beginning. For this purpose, three different case studies are used to show how concrete instances of the Omni Model look like. In particular, the Integration Model at the end of each case study covers the main aspects of the modeling concept, which is reviewed in a concluding overall discussion.

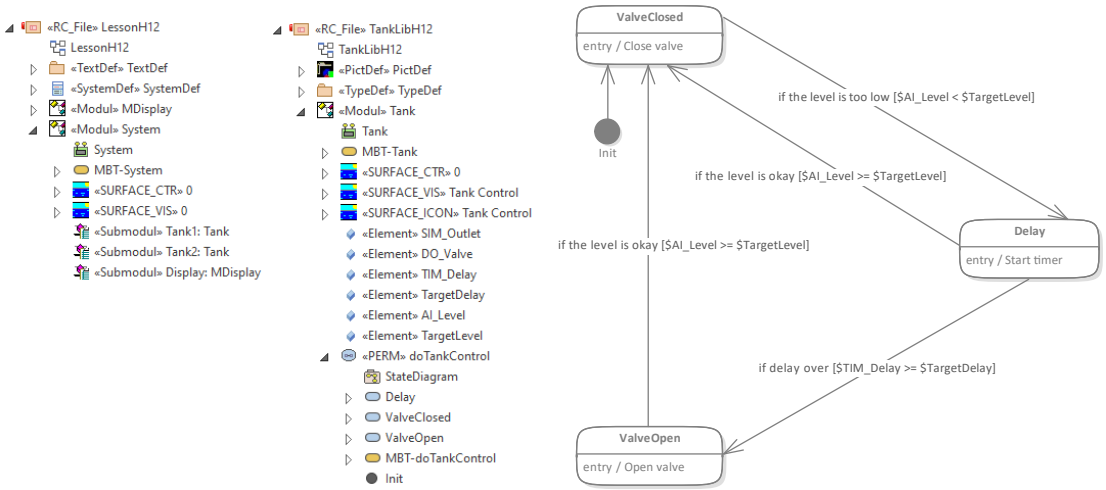
12.1 Tank Control System

The first case study represents a system that is used to visualize and control the level of liquid tanks. The system is designed in such a way that theoretically any number of tanks can be managed since the tank functionality is encapsulated. The functionality is realized via a tank unit that implements the level control. A tank comprises several physical components, including a level sensor and a valve for emptying the tank. The logic of the tank unit is represented at its core by a statemachine, which is picked up again in the next paragraph. Depending on the number of tanks managed, a corresponding number of instances of the tank unit are realized. In addition to the tank unit, the system has a display unit that shows all information about the tanks, such as the current fill level, the target fill level, or the status of the valve. The Omni Model of the Tank Control System presented in the following is used in particular in the context of section 13.1 to evaluate the Test Case Management functionality.

System Model

The actual development of the described system is done in *radCase*, which is a domain-specific modeling tool for embedded systems. The modeling language of the same name, which is developed by the company *IMACS*, allows systems to be implemented as a collection of many reusable function modules and is presented in section 7.1.1. These function modules can implement only functionality, but also contain components for visualization or documentation.

The System Model of the Tank Control System is shown in figure 12.1a. In particular, the left side shows the components of the `System` module, which includes the three submodules `Tank1`, `Tank2`, and `Display`. This instantiates the respective modules `Tank` and `MDisplay`. All in all, the radCase model comprises far more modules, which in particular realize the hardware-related software components and drivers. The model components of the `Tank` Module can be seen on the right side of figure 12.1a. The so-called `Elements` describe the internally used variables, which in particular define the state of the `Tank` module. For example, `TargetLevel` describes the target level that can be specified by the user.



(a) System Model Structure (b) Tank Control Statemachine

Figure 12.1: Tank Control System Model

The functionality of the `Tank` module is realized by the statemachine shown in figure 12.1b. This opens or closes the valve according to the target level, with a delay state preventing permanent actuation of the valve.

Test Model

A Test Model is specified for the presented System Model. For this purpose, the Test Modeling language of mbtSuite explained in section 7.1.2 is used. As already seen in figure 12.1a, a Test Model is specified based on the integration levels of the System Model (see for example `MBT-System` or `MBT-Tank`). These Test Models describe the expected behavior at the respective integration level and form the basis for the generation of concrete test cases. Figure 12.2 shows the structure of the Test Model `MBT-doTankControl` for the statemachine specified in the System Model.

The shown Test Model only results in some exemplary test cases and does not claim to provide complete coverage concerning a specific metric. The focus is on the demonstration of the concepts of the individual process steps of the MCSTLC, whereby the

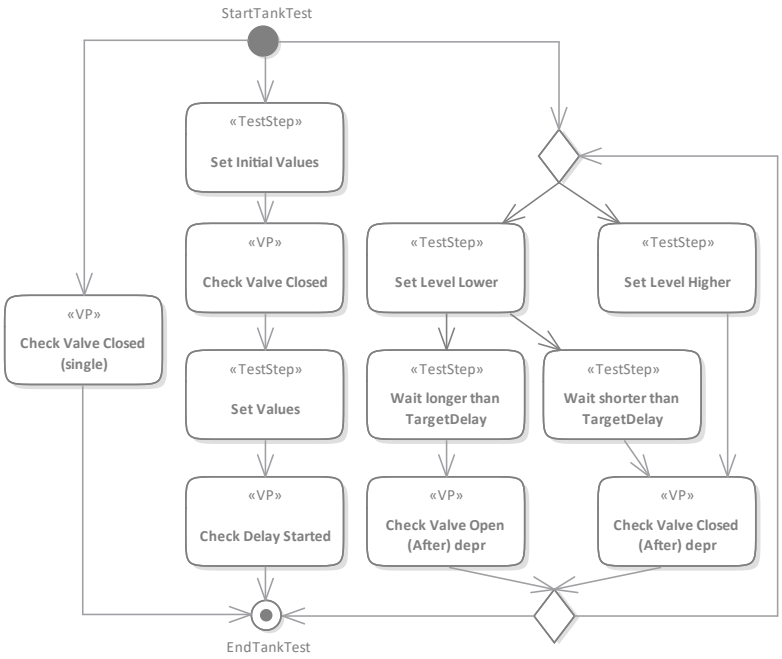


Figure 12.2: Test Model for Tank Control statemachine (`MBT-doTankControl`)

scalability of the individual partial approaches and the practical applicability can be concluded.

Requirements Model

Another modeling domain, which is part of the Omni Model of the Tank Control System Case Study, is the Requirements Model. For this purpose, the modeling capabilities of the radCase metamodel were used, which represents the requirements in a simplified modeling variant of the SysML requirements. Here, natural language requirements are utilized, which can be related using connectors. The concrete requirements are not discussed, but references are made to their use in the context of the Integration Model (see figure 12.3).

Integration Model

The most important model in the context of the Integrated Model Basis is the Integration Model. The modeling capabilities of the Integration Model have already been shown in section 7.1.3 and illustrated for the first time in the Running Example. Figure 12.3 gives an overview of the general structure of the Integration Model, as well as the structural links to model elements of connected modeling domains.

The gray model elements show the instantiated tree structure consisting of `IMComponents`

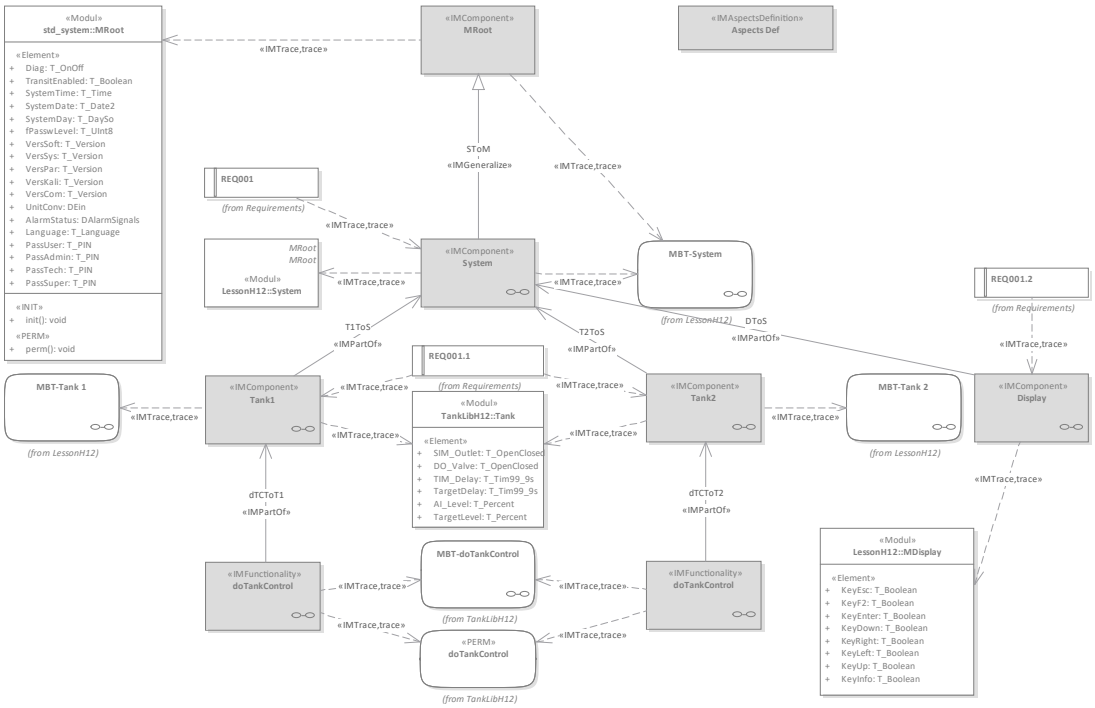


Figure 12.3: General structure of the Integration Model

and `IMFunctionalities`. An exception is the gray model element at the top right of the figure, which represents the `IMAspectsDefinition` of the use case. The aspects defined in it, which can be used in the context of the Integration Model to implement the Test Case Management mechanism, look as follows:

```
productline:String:set ['141414','141401'];
risklevel:Integer:ranged [1,10];
testprioritylevel:Integer:ranged [1,10];
encryptionbitcount:String:set ['256','512','1024'];
reqname:linked requirements:REQRequirement:name;
```

Next to the model elements of the Integration Model, the linked model elements of other modeling domains can be seen. Through the `IMTrace` links the connections between the Integration Model elements and model elements of other domains are specified. This level primarily describes structural relationships that specify corresponding model elements. In contrast, figure 12.4 specifies a subset of the Integration Model of the `IMFunctionality doTankControl`.

In particular, the corresponding elements of the behavior descriptions are linked. In this case, elements of the statemachine `doTankControl` and parts of the Test Model `MBT-doTankControl`. These links are utilized in the context of Abstract Test Execution. As seen in the previous sections, the individual modeling domains can be processed

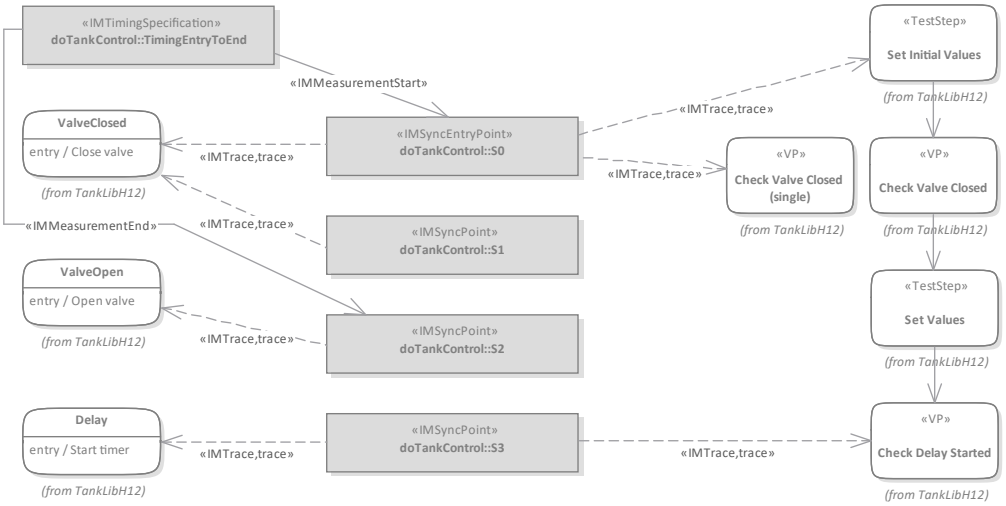


Figure 12.4: Model data embedded in the `IMFunctionality` `doTankControl`

as usual in the environment. The Integration Model links this heterogeneous model landscape to enable the analyses of the respective MCSTLC steps for the tank control system.

12.2 Automotive Light Control System

The second case study is a benchmark model developed by Peleska et al. that provides comprehensive modeling of a lighting control system in an automotive context [137][136]. The model includes the various control components that are integrated into the overall system and communicate with other components via different communication channels. The control logic comprises the following core components:

- `NormalAndEmerFlashing`: Controls the turn and emergency indication functionality and its dependencies
- `OpenCloseFlashing`: Manages the flashing related to the cars locking mechanism
- `CrashFlashing`: Controls the flashing behavior in case of a crash scenario indicated by the respective sensor
- `TheftFlashing`: Manages the flashing in case of unauthorized access to the car, which is indicated by the alarm system

In addition to these components, the present model includes a large number of components that regulate the exchange of information and are necessary due to the practical implementation of the corresponding hardware. These components are not in the focus of this presentation of the Omni Model. In particular, the focus is put on the components `CrashFlashing` and `TheftFlashing`, since they are used in the context of the evaluation of the Abstract Test Execution approach (see section 13.3).

System Model

To implement the functionalities mentioned, Peleska et al. use the modeling language UML. The modeling possibilities are very extensive so that in the context of our processing chain only the necessary information is represented by our simplified metamodel (see section 7.1.1). Figure 12.5 shows the high-level architecture of the described functionality.

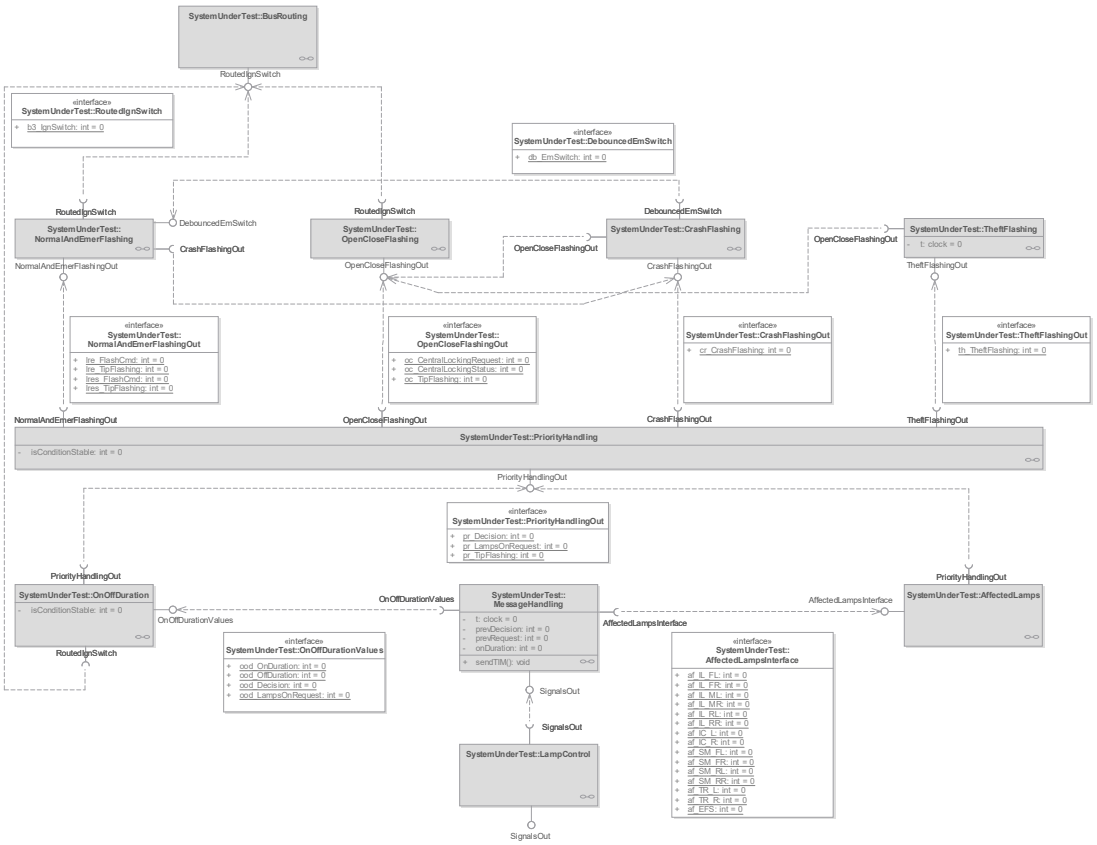


Figure 12.5: High-level architecture of Automotive Light Control System (ALCS) [136]

The gray model elements each represent parts of the overall functionality, which on the one hand contain the implementation of the partial functionality and on the other hand exchange information via interfaces. In addition to the model elements that concern the functionalities perceivable by the user, the `BusRouting` or the `PriorityHandling`, for example, is explicitly implemented by a separate component. To give an insight into the modeling of the partial functionalities, the implementation of the `CrashFlashing` and the `TheftFlashing` is shown in figure 12.6 and figure 12.7.

`CrashFlashing` is implemented by five different states. The system is switched from passive mode to active mode through bus signals, with intermediate states ensuring correct operation in the context of the present platform.

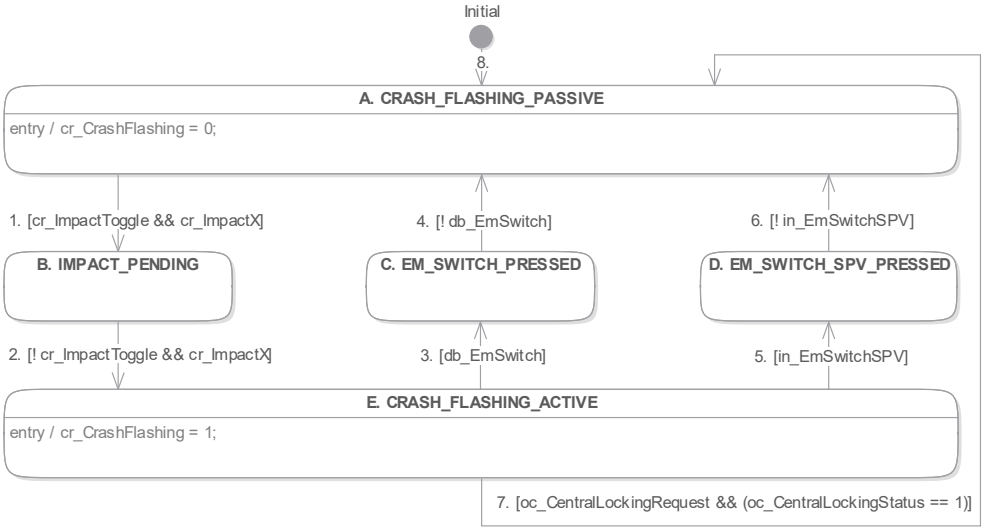


Figure 12.6: System Model for the `CrashFlashing` Functionality [136]

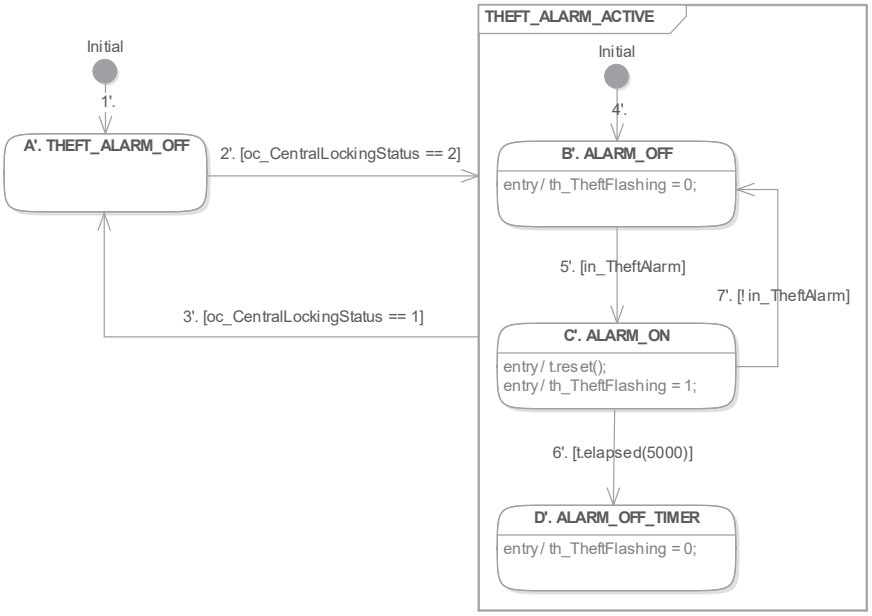


Figure 12.7: System Model for the `TheftFlashing` Functionality [136]

Similarly, the `TheftFlashing` is modeled by a statemachine. This hierarchical statemachine first distinguishes between an active and a passive mode. Provided that an unauthorized entry is detected by the alarm system (`in_TheftAlarm`), signaling is activated via the lighting system. This is deactivated if a timer has expired or the all-clear is given by the alarm system.

Overall, the two functional modules introduced to represent a small part of the overall functionality. In the context of the evaluation in section 13.3, these model parts are

chosen as representatives, whereby the knowledge gained can be transferred to the other system model parts. The prefixes of the transitions (`<number>.` and `<number>'.`) and states (`<letter>.` and `<letter>'.`) of the respective System Models are used as identifiers in the evaluation.

Test Model

Test Modeling in the context of the ALCS is realized based on the mbtSuite modeling language (see section 7.1.2). For this purpose, Test Models are created across the different integration levels to check the functionality specified in the System Model. Based on the System Model parts, `CrashFlashing` and `TheftFlashing` used for the evaluation, the corresponding Test Models are discussed in the further course.

Figure 12.8 shows the Test Model for the `CrashFlashing` functionality. It should be noted that the Test Model does not claim to be complete in terms of any coverage metric. Only a set of interesting test cases is mapped, which partially checks the functionality in edge cases. For this purpose, the necessary combinatorics of input parameters is determined systematically.

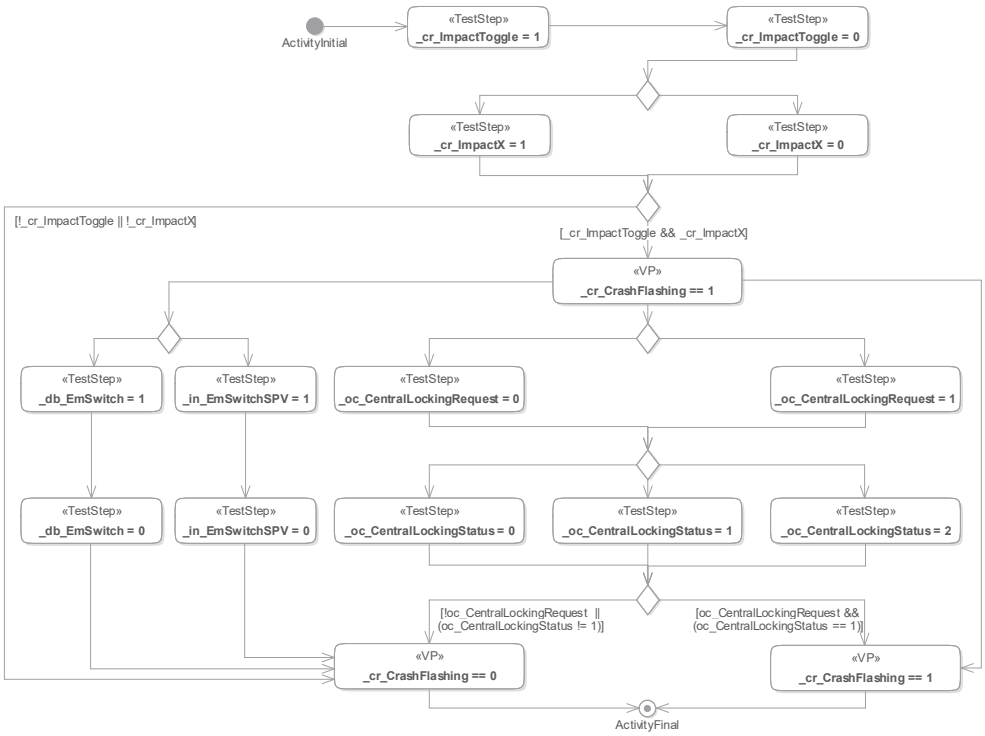


Figure 12.8: Test Model for the `doCtrlCrashFlashing` functionality

The same applies to the Test Model for the `TheftFlashing` functionality. Temporal considerations cannot be included at this level of abstraction, although the System Model specifies time-dependent behavior. Such tests are reserved for development phases

that can provide meaningful statements about the temporal behavior of the respective component and thus out of scope for this Test Model.

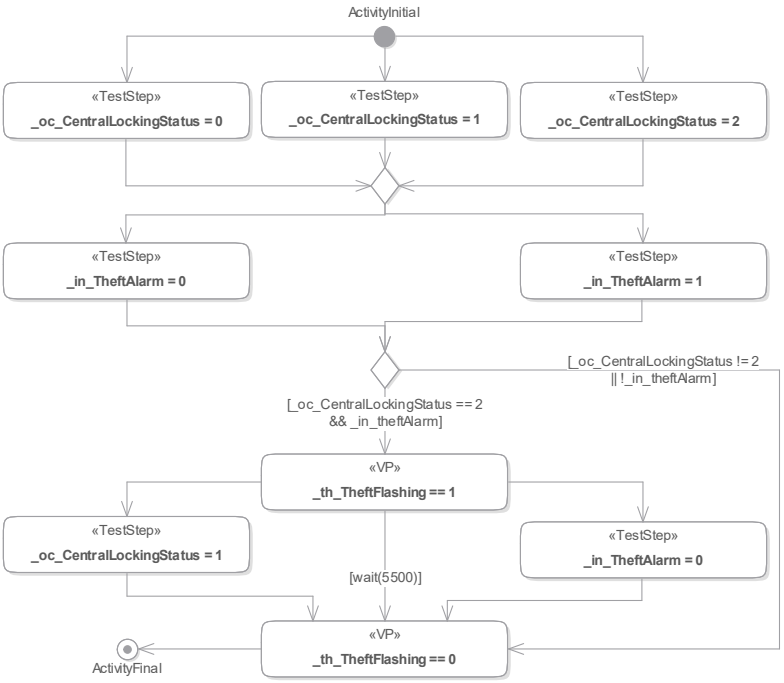


Figure 12.9: Test Model for the `doTheftFlashing` functionality

Requirements Model

Another modeling domain that is covered in this case study is the Requirements Model. For the ALCS, however, we did not rely on purely natural-language requirements modeling, but on modeling using UML UseCase diagrams. For this purpose, a simplified metamodel is created, which represents the essential model information for our context. However, it must be emphasized that there is no limitation of our approach, but is only done for clarity and manageability.

Figure 12.10 shows a snippet of the modeled UseCases and how they are connected to the actors in the system context.

Internally, the scope of the mapped UseCases is still refined using natural language requirements. In the process, consistent naming of concepts is usually introduced, particularly in the functional requirements, which is partially reflected in the naming of signals of the System Model.

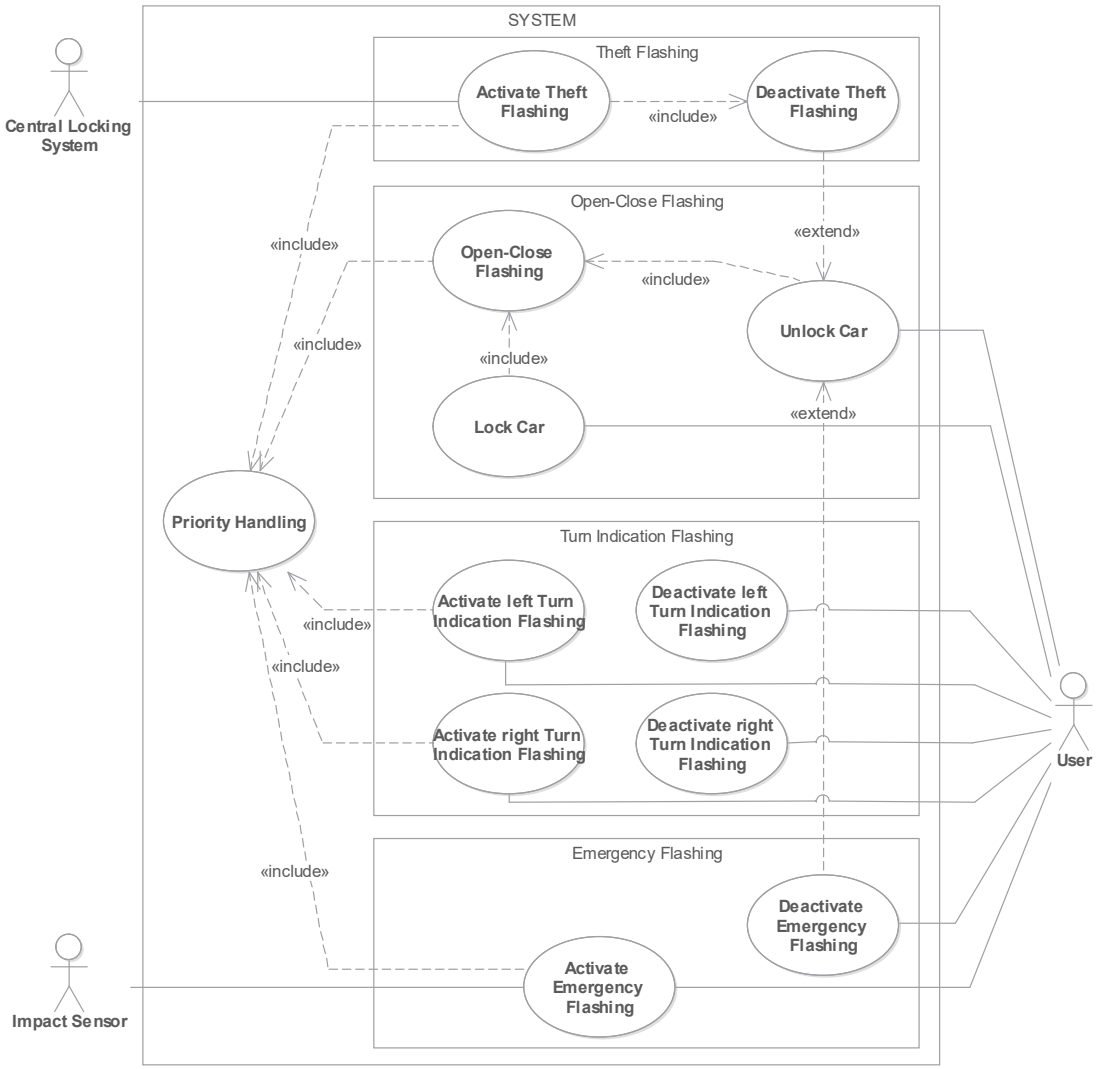


Figure 12.10: Requirements Model for the ALCS

Integration Model

Finally, we discuss the most important modeling domain, the Integration Model of the ALCS. Since the described system contains a large number of functionalities, which are encapsulated and interconnected via several integration levels, the Integration Model is significantly larger than the respective artifact of the Tank Control System from the previous section. Figure 16.19 (Supplementary Material) gives an impression of this but does not claim to provide a comprehensive model. The gray components represent the structural model elements of the Integration Model, while the white components represent connected model elements of other modeling domains. Figure 12.11 shows an excerpt of this comprehensive Integration Model, which concerns the excerpt of `Crash-Flashing` focused on in advance.

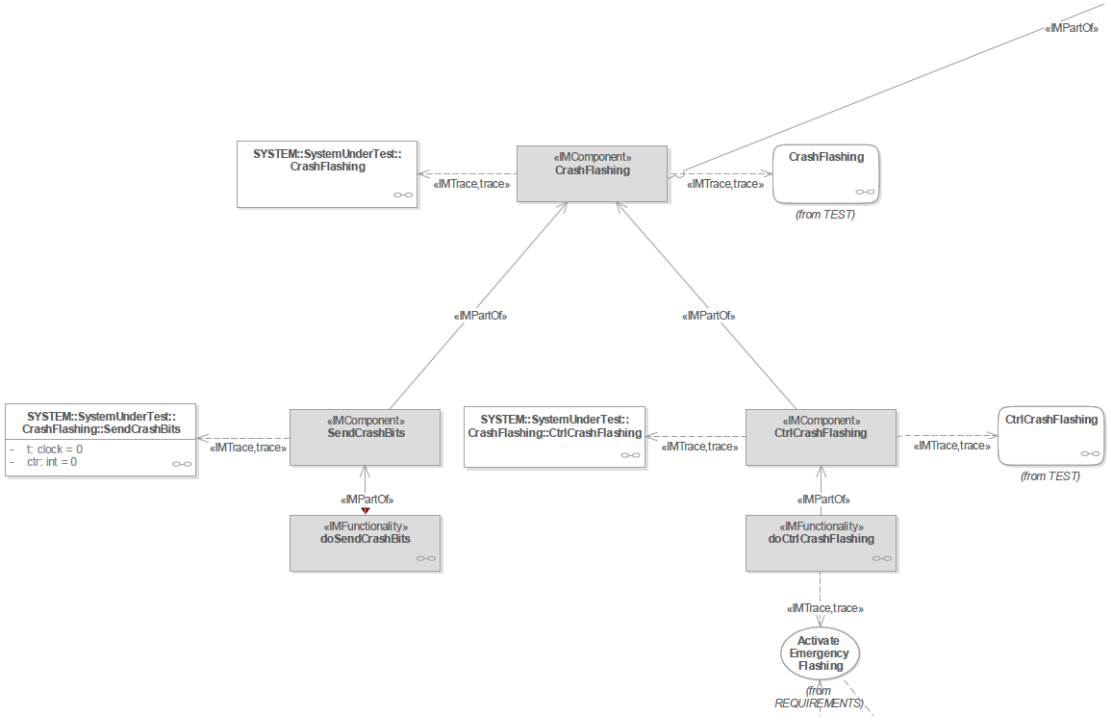


Figure 12.11: Excerpt of figure 16.19 focusing `CrashFlashing`-related parts

On the one hand, this figure shows how model elements of System Modeling and Test Modeling are linked across the different levels of the Integration Model employing `IMTrace` connectors. Furthermore, a linked use case from the respective Model can be seen at the bottom of the figure. If there were other modeling domains, such as safety or security models, they could be linked respectively and provide valuable information for the MCSTLC built on top.

Figure 12.12 shows the models used in the `IMFunctionality doCrtlCrashFlashing` stored mappings of behavioral models or their model elements.

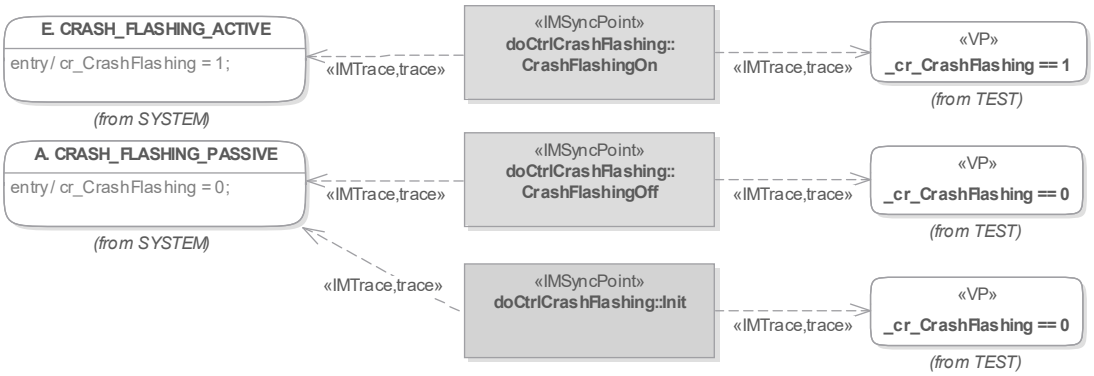


Figure 12.12: Internals of the `IMFunctionality doCtrlCrashFlashing`

Similarly, figure 12.13 shows the links for the corresponding model elements in the context of `IMFunctionality doTheftFlashing`.

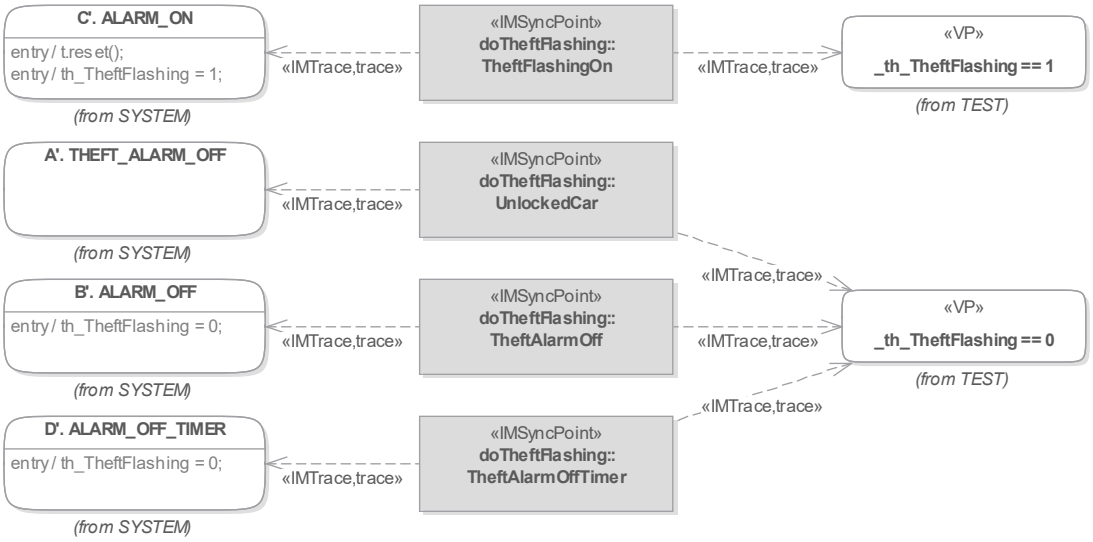


Figure 12.13: Internals of the `IMFunctionality doTheftFlashing`

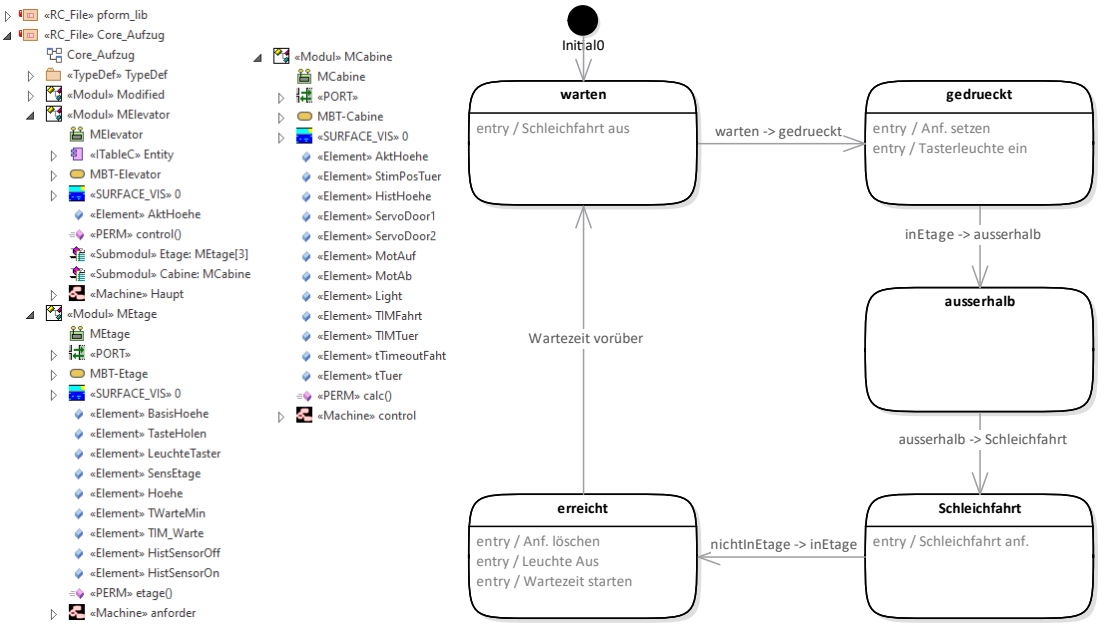
In addition to the structural model elements of the Integration Model, the `IMAspects-Definition` is included, which defines the valid Aspects in the context of this Integration Model. The concrete characteristics of the Aspects are not discussed in detail, since they do not provide any new insights.

12.3 Elevator System

The last case study deals with an elevator system. The system consists of a cabin that can move over three floors. One feature of the elevator system is a control panel placed in the cabin that allows the passenger to select the destination floor. The floor, on the other hand, has a button that enables the passenger to initiate an elevator request on the floor in question. The specific functionality becomes clearer in the System Model which is described in the following section. The Omni Model presented throughout the rest of this chapter with all its modeling domains is primarily used in the context of the evaluation of the Mutation Analysis approach in section 13.4.

System Model

The system has again been created in the context of the modeling tool *radCase* and its homonymous modeling language. The model is divided into the concepts `MElevator`, `MEtage`, and `MCabine`. An overview of the parts of each component can be seen in figure 12.14a.



(a) System Model structure

(b) Statemachine managing the floor requests (`anforder`)

Figure 12.14: Elevator System Model

`MElevator` represents the parent concept and contains three instances of `MEtage` and one instance of `MCabine`. Furthermore, a statemachine `main` and a method `control` manage and coordinate the functionality.

`MEtage` maps the addressed functionality of the respective floor. In particular, the functionality of the control panel is handled, as well as the height control and the waiting times. The functionality of the floor is again realized by a combination of a C-function `etage` and a statemachine `request`, whose structure can be seen in figure 12.14b. Based on this snippet of the System Model, the Mutation Analysis functionality is checked in the further course of the evaluation.

`MCabine` encapsulates the concepts that are assigned to the elevator cabin in the context of this System Model. In addition to some variables that primarily control the mechanical components, the functionalities are represented by a combination of a C-function `calc` and a statemachine `control`.

In addition to these components, which are purely limited to the elevator functionality, a very large number of radCase libraries are included in the model (see figure 12.14a above). These libraries are responsible for the control of the hardware, the communication functionality, and other functionalities of the overall system, but are out of scope for our use case.

Test Model

As already seen in figure 12.14a, the Test Model relies on the modeling capabilities of the `mbtSuite`. The Test Model is hierarchically structured and thus covers different integration levels of the system at hand. At the system test level, the Test Model `MBT-Elevator` ensures that the interaction of the encapsulated components is implemented according to the requirements. Accordingly, on the integration test level, the Test Models `MBT-Cabine` and `MBT-Etage` cover the emergent functionality of the system parts, which especially concerns the interaction of the algorithms. For unit tests, the Test Model for the statemachine `anforder` mapped in figure 12.15 is detailed in this context.

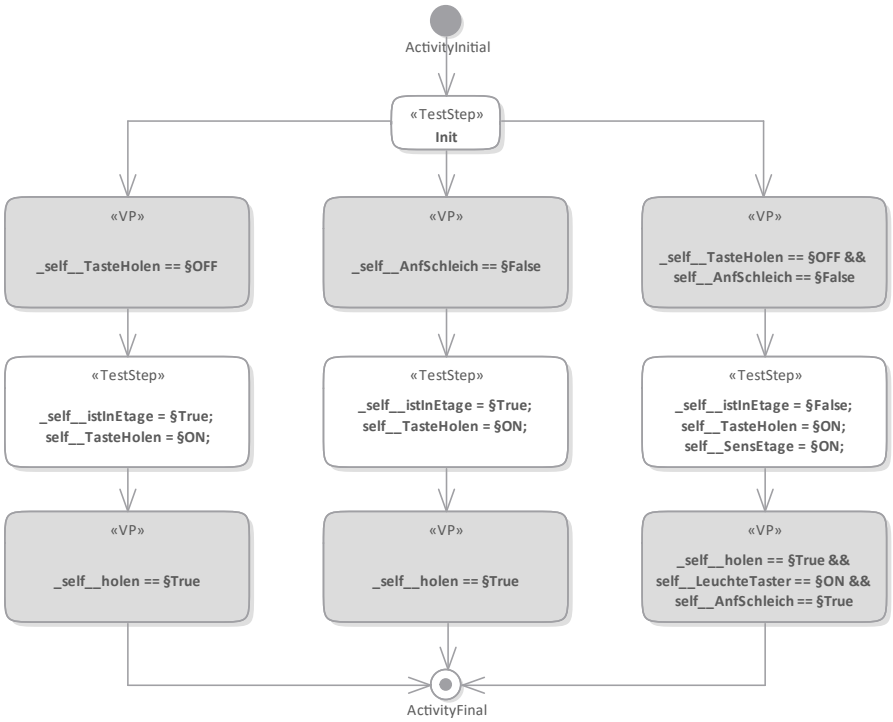


Figure 12.15: Scoped Test Model for the `anforder` System Model

This simple model, which results in three test cases, is used in the context of evaluating the Mutation Analysis approach. The completeness concerning common coverage measures is not achieved at this point either, but this is not the focus of this case study.

Fault Tree Model

In contrast to the case studies shown so far, the elevator system does not explicitly address the Requirement Model. Instead, a fault tree from the safety modeling domain is included in this Omni Model. To be able to map such model information, a metamodel is designed that enables the specification of fault trees (see part VI (Supplementary Material)). In the modeling scope, all common `Event` and `Gate` types of a fault tree are implemented.

Figure 12.16 shows a fault tree that maps possible faults, the occurrence probabilities, and their propagation for the presented System Model in a structured way.

In the model, mainly hardware-specific faults are considered, which can be related to the software-side components as well. In the context of the Omni Model, this model is used in particular for targeted and risk-sensitive testing. Here, the occurrence probabilities stored in the events are evaluated mainly.

Integration Model

Finally, the Integration Model created for the Elevator System is considered. At this point, the modeling capabilities of the Integration Metamodel presented in section 7.1.3 are used to implement the relationships between modeling domains as well as the metainformation. Since the complete Integration Model of the Elevator System is relatively extensive, an overview is shown in figure 16.18 (Supplementary Material), but the detailed information is no longer readable. An excerpt of the Integration Model can be seen in figure 12.17, where the focus is again on the components that are used in the course of ongoing evaluation work.

In particular, the concepts on the part of the Integration Model are shown, which correspond to the three instances of `MEtage` of the System Model. These are structurally linked to the corresponding Test Model instances `MBT-Etage[0-2]` and the System Model definition `MEtage`. The same applies to the functional components `anfoder` and `etage`, which are linked to the corresponding model elements of the System and Test Model, respectively.

Again, if we consider the model components that are stored in the `IMFunctionality anforder`, the modeling shown in figure 12.18 can be seen.

Here, the states of the corresponding System Model are connected to the corresponding `VPs` of the Test Model. These connections are in turn implemented by using the `IMSyncPoint` concept. Specifically, the `IMSyncPoints SP1` through `SP3` are pronounced. Further meta-information is not implemented in this context, since it has no relevance for the use case but can be implemented analogously to the modeling in the context of the other case studies.

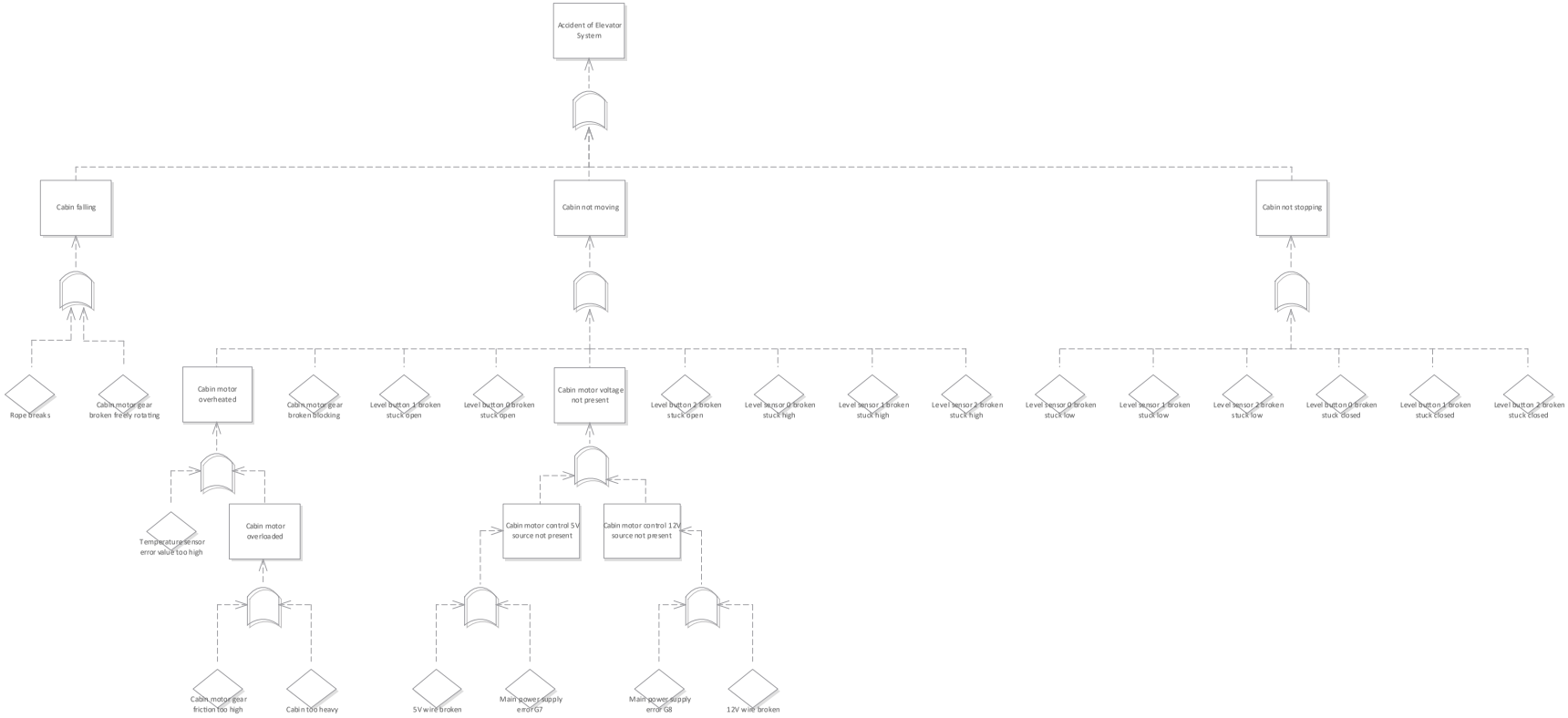


Figure 12.16: Fault Tree Model for the Elevator System

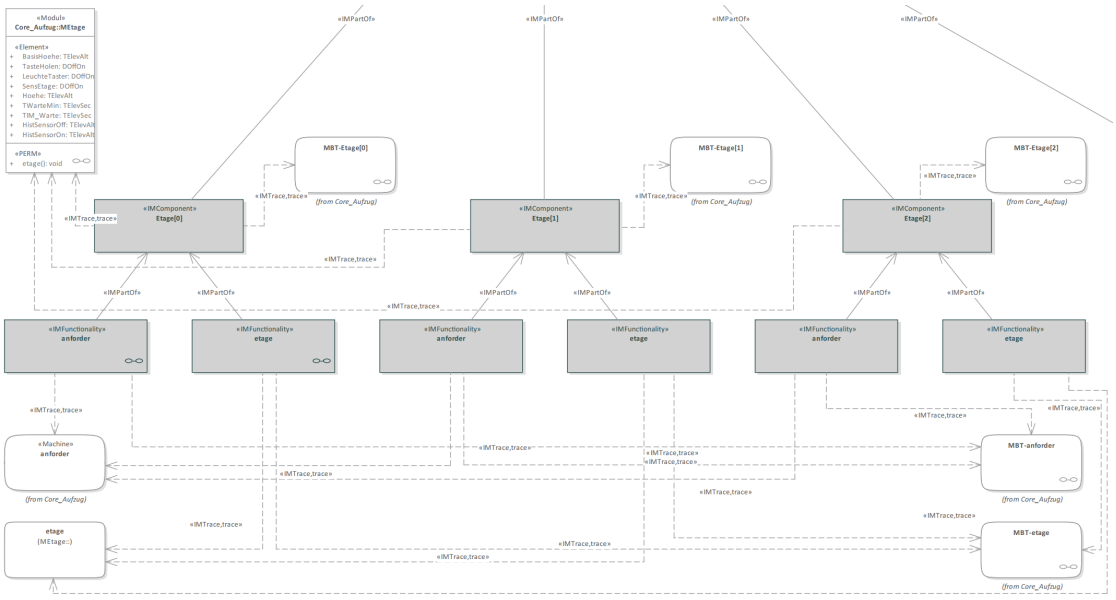


Figure 12.17: Excerpt of the Integration Model focusing the three instances of `MEtage`

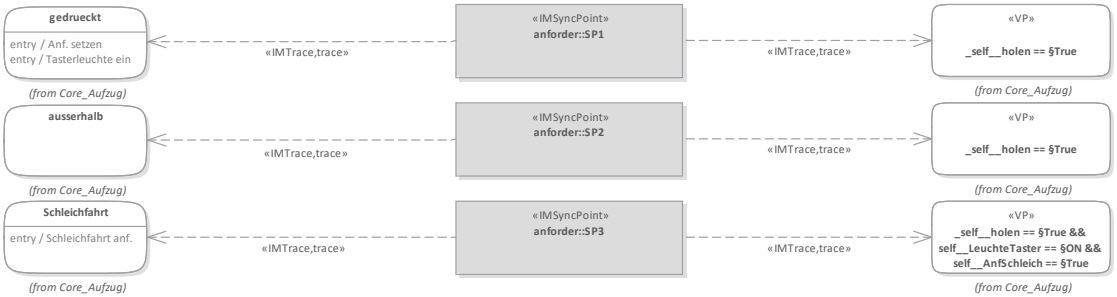


Figure 12.18: Behavior mapping information of the Integration Model for the `anforder` System Model

12.4 Discussion

Through the presented case studies *Tank Control System*, *Automotive Light Control System*, and `Elevator System`, the application of a variety of modeling possibilities of the conceptual representations from part III was shown. This demonstrates the flexibility and extensibility of the Omni Model approach. In terms of a qualitative evaluation of the Omni Model concept, aspects such as *Adaptability*, *Scalability / Extensibility*, *Modularity / Reusability*, and *Maintainability* are explicitly discussed in the course of this section about the case studies.

Adaptability

This aspect primarily considers the ability of the approach to be adapted to the application context and the requirements of the concrete use case. The adaptability of the Omni Model concept is a central component and is essentially achieved through the interchangeability of the modeling languages of the individual modeling domains, the model transformations of the use case-specific metamodel to the internal representation, as well as the link and meta information in the Integration Model. Due to the adaptability of the artifacts, the approach can be adapted to almost any model-based development context. The model transformation step allows the expert to correct incomplete semantics in the source metamodels or to adapt it to other components of the development process, such as code generators. Furthermore, the specification of the structural relationships between the System and Test Model inside the Integration Model enables heterogeneous modeling and development approaches to be adapted. The meta-information stored in the Integration Model can be used to adapt to the established development methods of the application context and the development practice implemented in the company.

In the context of the case studies and the running example, two of the three mentioned aspects are demonstrated by combining different modeling languages (UML, SYSML, radCase, mbtSuite, UTP). Only the aspect of model transformation is not explicitly shown, but is clarified in the context of the following evaluations of the process steps of MCSTLC.

Scalability / Extensibility

In the context of this aspect, the scalability and extensibility of the Omni Model are evaluated, where this is discussed for the instantiation as well as the underlying concepts. The former is not limited from a theoretical point of view, i.e. the concept can be scaled arbitrarily on the part of the model instances. The most comprehensive case study presented is the ALCS (see section 12.2). In this case study, it turns out that the scalability of any involved modeling domains remains unchanged compared to non-Omni Model approaches. Only the newly introduced model artifact, the Integration Model, becomes

confusing in its current implementation for very extensive Omni Models. However, this aspect can be remedied with suitable tool support, for example by collapsing subtrees of the Integration Model.

Extensibility is also provided from the point of view of the overall concept. For example, theoretically, any number of modeling domains can be included in the Omni Model. The same applies to the modeling of meta-information via the Aspects concept. Here, any number of Aspects can be defined and expressed on the model elements of the Integration Model.

The case studies and the running example do not cover very large Omni Models. However, if the presented Omni Model instances are considered in combination, the mentioned properties are tangible. Only the ALCS gives an impression of the aspects of scalability and extensibility.

Modularity / Reusability

The modularity of the Omni Model concept can be seen in the context of the adaptability mentioned at the beginning. The fact that different modeling domains are synchronized via the Integration Model, demonstrates the modularity of the concept in general. During model-centric development, however, the Omni Model approach promotes the expression of modules in the software or system architecture. This modularity across domains creates modules that link related model parts of all involved domains. Examples of such modularization of the developed overall system can be seen in the case studies and their sections on the Integration Model (e.g. see section 12.2).

Building on this insight, the second aspect of this section, reusability, is motivated. The Omni Model approach is reusable by design since the processing chain is detached from the original modeling artifacts. Considering reusability on the concrete Omni Model instance, the approach generally leads to more easily reusable software, such that the modules just addressed from model artifacts of different domains can potentially be reused in a different application context. This is particularly the case if the modeling guidelines of the respective modeling languages are heeded to reduce emergent effects and to guarantee a design for testability (see for example section 7.1.1). During the application of the Omni Model approach, especially during the implementation of the respective Integration Model instances, this effect could be confirmed across all case studies by the developers involved.

Maintainability

The last aspect we discuss in the context of the qualitative evaluation of the Omni Model approach is maintainability. Because the Omni Model approach largely leaves the original artifacts of the individual modeling domains unchanged and places the

separation of concerns in the foreground, the maintainability of these models is not negatively affected. The structural and behavioral links between the information tend to make changes and their impact easier to understand. This can be supported by appropriate tool support. However, a well-maintained Integration Model is a prerequisite at this point, which means that the critical component concerning maintainability is given. In manageable project sizes, maintainability is not a problem, but merely a question of the distribution of responsibilities. If the models become larger, as is indicated in section 12.2, for example, sufficient tool support is necessary.

Another dimension to be considered in the context of maintainability are the model transformations to the respective modeling languages. In the case of a change or extension of the modeling capabilities, the changes have to be made consistently with the existing implementation of the transformation. Otherwise, the usability of the results of all processing steps of the MCSTLC based on them can no longer be guaranteed.

13

Qualitative and Quantitative Evaluation of the MCSTLC Approaches

In this chapter, the process steps based on the Omni Model are evaluated. For this purpose, the case studies presented in the previous chapter, as well as synthetically generated example models are used to enable a quantitative evaluation. Some aspects are subjected to a qualitative evaluation, whereby the evaluation setup is presented at the beginning of each section before the evaluation is carried out.

13.1 Model-based Test Case Management

In this section, the Test Case Management approach is qualitatively evaluated. In particular, the Tank Control System introduced in section 12.1 is used for this purpose.

Evaluation Setup

As part of the evaluation, the following steps are performed (see figure 13.1) to examine the developed Test Case Management concept.

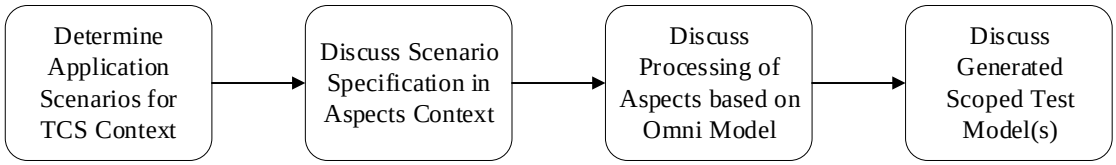


Figure 13.1: Core aspects of evaluation

In the first step, three test scenarios are specified based on the defined aspects, which encode different test intentions of the expert in the form of aspect constraints. These scenarios cover in particular special cases of the concept and demonstrate the functionality in practical application. In the second step, the introduced scenarios are transformed into a specification according to the `IMAspectDefinition` of the Tank Control System. The transformation of the natural language specified test focus of the selected

scenarios to an aspect constraint as well as the mapping of the intended test levels is discussed. Furthermore, the quantifiability of the contents is briefly considered in this context. As a result of this step, the created aspect constraint snippets of the scenarios are shown, which form the basis for the next step of the evaluation. In the third step, the evaluation of the Aspect Constraint Snippets of the scenarios, based on the model elements of the Integration Model (see figure 12.3), is shown. Here, the `IMAspectSpecifications` of the Integration Model, as well as information of linked model elements of other modeling domains are evaluated. In the last step, it is shown how the resulting Scoped Test Models are derived from the reduced Integration Models in the context of the Tank Control System. Finally, the processing of the selected scenarios and the results are discussed concerning qualitative aspects such as scalability.

The following scenarios are used for the evaluation:

1. As part of the company's requirements-driven test strategy, the next step is to test the functional units and their integrations specified in the context of requirement `REQ001.1`. In particular, a risk-sensitive approach has to be implemented, whereby components with a risk assessment of less than `2` are to be left out at first.
2. Within the scope of the integration tests of the Tank Control System, all functional components of any product line that have received a prioritization value greater than `7` concerning test planning are to be tested.
3. As part of the GUI testing of the Tank Control System, all components of the system that are related to requirement `REQ001.2` and a risk assessment value below or equal to `2` shall be tested.

The following representation of the Integration Model (figure 12.3) focuses on the included `IMAspectSpecifications` of the model elements and is used as a basis in the further course of the evaluation.

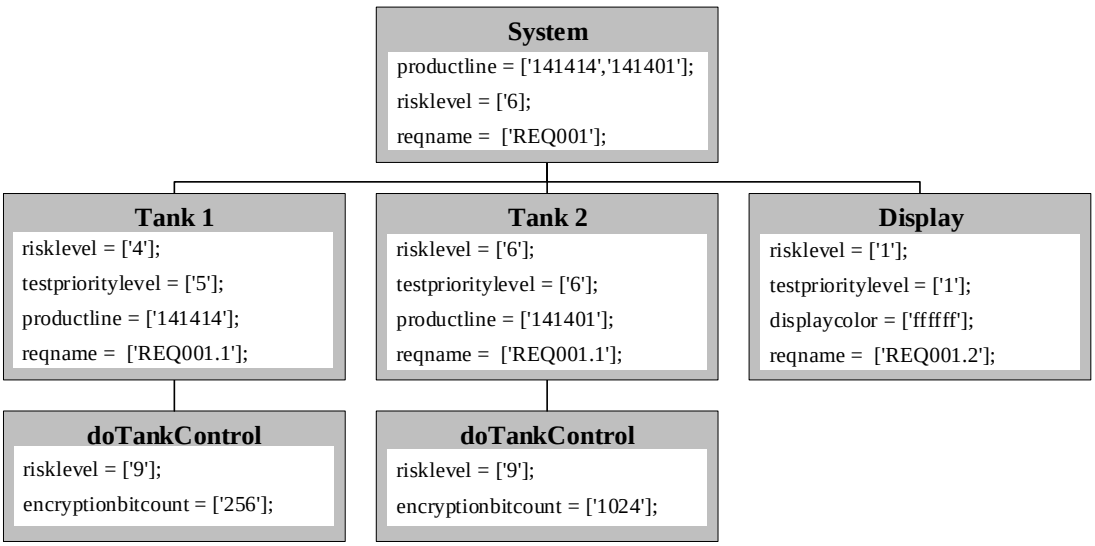


Figure 13.2: General structure of the Integration Model with the Aspects focused

At this point, the intrinsic and the synthetic aspects are represented in the same way.

Evaluation

In this evaluation, the steps for one scenario are presented in their entirety before the next scenario is discussed. The Aspects shown in section 12.1 are available for the implementation of the Aspect Constraints. According to the described procedure, therefore, the natural language description of `scenario 1` is first converted into an Aspect Constraint as well as test level constraint. First, the intended test level is derived from the text passage "[...] the functional units and their integrations [...]", namely both the `unit` and the `integration` test levels. Furthermore, the focus is explicitly put on components of the system that are related to requirement `REQ001.1`. Therefore, the first part of the Aspect Constraint is specified by restricting the intrinsic Aspect `reqname` accordingly. The second part of the Aspect Constraint is determined by the text passage "[...] components with a risk assessment of less than 2 are to be left out at first". This results in the second constraint, which is based on the synthetic Aspect `risklevel`. Overall, the following parameters are determined for the prototypical implementation of the scoping functionality:

```
aspectconstraints = reqname:in ['REQ001.1'] & risklevel:gre ['2'];
                  testlevels = 'unit,integration';
```

In this case, the quantification of the thresholds for the Aspect Constraints is clearly derivable, but this can be challenging in some cases.

In the next step of the Test Case Management concept, the Aspect Constraint is applied to the Integration Model at hand. According to the described procedure, we start with the root node of the Integration Model and iteratively evaluate the model elements about the fulfillment of the Aspect Constraint. `Scenario 1` results in a reduced integration model consisting of the model elements `Tank 1`, `Tank 2`, and their respective `doTankControl` functionalities.

Based on the reduced Integration Model, the links to Test Models according to section 12.1 and the specified test levels, the set of relevant Scoped Test Models is determined. Specifically, these are the Scoped Test Models `MBT-Tank 1`, `MBT-Tank 2` and `MBT-doTankControl`. The fact that no unique Scoped Test Model has emerged in this case offers the possibility to use the Test Case Management approach by iteratively increasing the limit of the `risklevel` to its maximum. In this way, the order of test execution can be fine-tuned along with the risk assessment values.

After evaluating the implementation of `scenario 1` in the context of the tank control system, we now examine `scenario 2`. This scenario explicitly focuses on integration tests, hence this test level is specified in our configuration. First, the text passage "[...] components of any product line [...]" is relevant on the part of the Aspect Constraints,

specifying that test cases should be considered for all defined product lines. Second, the statement about prioritization based on test planning results in a constraint on the synthetic Aspect `testpriority`. Overall, the natural language description results in the following parameters for the prototypical implementation:

```
aspectconstraints = productline:in ['141414','141401'] &
                    testpriority:gr ['7'];
                  testlevels = 'integration';
```

In this scenario, the difficulty for the user is the required knowledge about the value ranges of the existing Aspects. For example, for the specification of the first part, any `IDs` of the product lines must be considered, which can be taken from the `IMAspect-Definition`. At this point, the language scope of the Aspect Constraint Language can be utilized and the corresponding logical counter-statement can be formulated.

Next, a reduced Integration Model is determined according to the parameters. The application of the first component of the Aspect Constraint, namely the restriction with respect to the product line considered does not result in any reduction. Due to the all-quantification of the range of values and the partial absence of specifications for this Aspect, the constraint is never evaluated negatively. The second part of the Aspect Constraint excludes the model elements `Tank 1`, `Tank 2`, and `Display` and therefore only the model elements `System` and `doTankControl` of each of the two tanks remain in the reduced Integration Model.

In the last step, the set of relevant Scoped Test Models is determined based on the information of the Integrated Model Basis. Based on the relation information, the two Test Models `MBT-System` and `MBT-doTankControl` represent the intermediate result. However, considering the second parameter, which restricts the test level, the empty set is the result of this Test Case Management evaluation. This scenario shows that the two-step application of constraints on separate model artifacts can cause an empty set only in the last processing step. While the evaluation is correct, it may not reflect the expert's intent. Evaluation using this scenario has shown the flexibility of the solution, but also its pitfalls.

Last, `scenario 3` is evaluated throughout the rest of this section. First, the text passage "[...] related to requirement `REQ001.2` [...]" indicates that part of the Aspect Constraint is based on the intrinsic Aspect `reqname`. Furthermore, the risk assessment of the model parts is again included, which is limited to values less than or equal to `2`. The intended test level is derived from the GUI test context in this scenario, resulting in the following parameters for the subsequent evaluation.

```
aspectconstraints = reqname:in ['REQ001.2'] & risklevel:lee ['2'];
                    testlevels = 'system';
```

Evaluating these constraints on the Integration Model of the Tank Control System results in a reduced Integration Model, which merely consists of the model element `Display`. Including the links to model elements of the Test Model as shown in figure 12.3, the resulting set of Scoped Test Models is the empty set. In this case, the reason is the absence or missing link of the corresponding Test Model. This scenario shows once again the importance of the Integration Model in the context of the Test Case Management concept. Errors in the specification or maintenance of this model artifact have a significant influence on the quality of the results.

Overall, the evaluation of the approach based on the example scenarios has shown that the developed concept for Test Case Management is promising. In terms of *Adaptability/Reusability*, the concept is flexible by definition and can be used in any application context. During the evaluation runs of the prototypical implementation, a positive impression could be gained on the part of *Scalability*. For example, across our case studies, much larger synthetically generated parameter sets are transferred to a set of Scoped Test Models on a consumer notebook within a few milliseconds. Regarding the *Maintainability* of the concept, there were likewise no negative points. Only the *Maintainability* of the Integration Model stands out as a weak point, but this cannot be attributed to the Test Case Management concept. The Test Case Management approach was applied and evaluated across the other case studies as well. The findings obtained are consistent with the facts presented in the course of this section and confirm the objectivity of the statements made.

13.2 Model-based Test Generation

In this section, the approach for generating test cases from Test Models explained in chapter 9 is evaluated. Therefore, a quantitative evaluation is performed, which highlights the time frame, the scalability, and the adaptation aspect of our approach. Besides, a qualitative evaluation and discussion of the remaining concepts are carried out.

Evaluation Setup

To be able to investigate the mentioned aspects of this processing step, a suitable evaluation setup needs to be selected. An overview of the sub-steps, as well as technical components of the evaluation, can be found in figure 13.3. In the first step, a module of our prototypical implementation of A3F uses an EGPP model generator to be able to generate arbitrary model instances. These EGPP instances are intended to represent the Scoped Test Models in the context of the evaluation, which usually emerges from the Test Case Management processing step. The model generator module (`egpp_generator`) provides a variety of configuration options that allow the user to influence the appearance of the generated model instance. The model generator creates

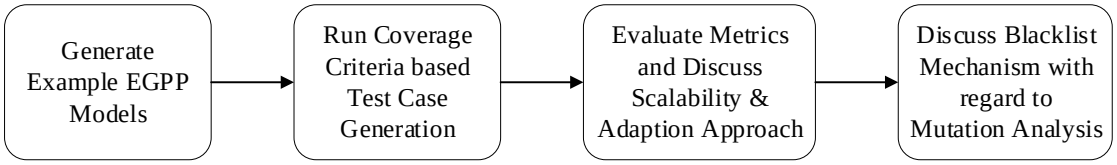


Figure 13.3: Core aspects of evaluation

the EGPP instances iteratively, starting from an `EGPPInitialNode`, by inserting different model components (branches, loops, subgraphs, ...) according to a probability of occurrence.

As can be seen in listing 13.1, these occurrence probabilities can be specified. In the context of performing the evaluation, two different configurations of the `egpp_generator` analysis are used, which produce instances as they occur in practical use cases. Furthermore, the two configurations are executed on two different performing platforms, respectively used to generate EGPP instances. On the one hand, a *Low Performance* platform (`Intel i7 2 Cores @ 3.6Ghz, 16Gb RAM`) and on the other hand, a *High Performance* platform PC (`AMD Ryzen 7 8 Cores @ 4.3Ghz, 16GB RAM`) are used to show the differences and at the same time the possibilities of different hardware. Regardless of the platform or configuration, instances of different sizes are generated by the generator, whereby the number of model elements of a classic Test Model is usually far below the model sizes chosen here.

```
1  analysis egpp_generator(egpp){
2    max_nodes="10";
3    max_depth="1";
4    dec_factor="4";
5    par_factor="1";
6    prob_node="0.5";
7    prob_dec="0.5";
8    prob_par="0.0";
9    prob_loop="0.0";
10   prob_sub="0.0";
11   prob_break="0.0";
12   taggeddata_on="0";
13 }
14 analysis egpp_path_generation(edge) {
15   integrationmodel="";
16   autoadvice="";
17   startcriterion="|edgecoverage|MAX_PATHS:1000";
18   testmodels="egpp_generator|egpp|DataTransformationResult|getOutputElements";
19   blacklistthreshold="0.0";
20 }
21 analysis egpp_path_generation(node) {
22   integrationmodel="";
23   autoadvice="";
24   startcriterion="|nodecoverage|MAX_PATHS:1000";
25   testmodels="egpp_generator|egpp|DataTransformationResult|getOutputElements";
26   blacklistthreshold="0.0";
27 }
28 analysis egpp_path_generation(path) {
29   integrationmodel="";
```

```
30   autoadvice="";
31   startcriterion="|pathcoverage|MAX_PATHS:1000";
32   testmodels="egpp_generator|egpp|DataTransformationResult|getOutputElements";
33   blacklistthreshold="0.0";
34 }
```

Listing 13.1: A3F configuration for the evaluation of Test Case Generation approach

In the second step, the generated EGPP instances are respectively processed by our A3F module to generate test cases according to *node*, *edge*, and *path coverage*. These purely structural coverage criteria do not take into account the annotated data, which usually determine the number of loop iterations, which is why generation is aborted for such cases when the number of `1000` test cases is determined.

In the third step, the data collected during automated processing is elaborated and suitably presented. It is shown how the processing time, as well as the number of test cases, behave depending on the model size. Furthermore, the average processing time is considered as a function of the model size, from which conclusions can be drawn regarding the scalability of the approach. Also, the empirically determined data are used to show the meaningfulness of the adaptation mechanism.

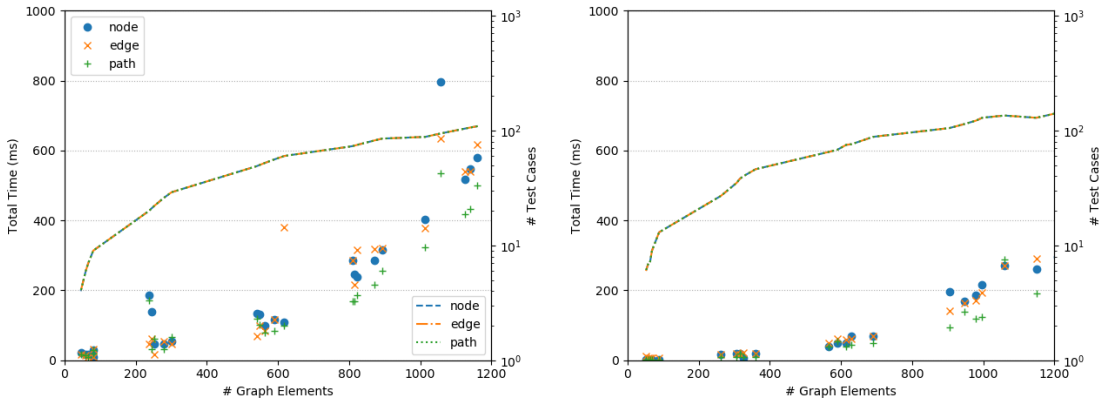
In the last step, the blacklisting mechanism based on the determined test case set is discussed. In particular, the interaction and the improvement potential of this extension are evaluated.

Evaluation

Besides running the experiments on different performing platforms (*Low* vs. *High Performance*), the selection of the generated EGPP instances is additionally chosen to be as diverse as possible to gain as much expressiveness as possible. As can be seen in figure 13.4, in each of the four subgraphs different combinations of *Platform* and *basic properties of EGPP Instances*(*No Loop*/*Loop*) are shown. In each case, the set of data represented by discrete measurement points represent the `Total Time (ms)` relative to the `Number of Graph Elements`, i.e. the absolute size of the EGPP instance. Here, the circle represents the measurements for the *Node Coverage Criterion*, the cross represents the measurements for the *Edge Coverage Criterion*, and the plus represents the measurements for the *Path Coverage Criterion*. Similarly, the continuum of measured values shows the relationship between the `Number of Test Cases` and the `Number of Graph Elements`. Again, a distinction is made between the known criteria. A dashed line represents the measured values of the *Node Coverage Criterion*, a dash-dotted line, in turn, represents the values of the *Edge Coverage Criterion*, and the dotted line is used for the *Path Coverage Criterion*.

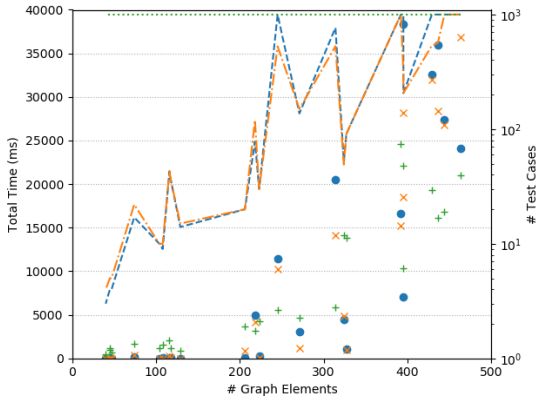
The evaluations in figure 13.4a and figure 13.4b show the measured values for EGPP Instances consisting of `10` to `1200` model elements. The proportions of each model element type behave according to the A3F configuration introduced in listing 13.1. In

contrast, for the evaluations in figure 13.4c and figure 13.4d, the said configuration is adapted. Essentially, the occurrence probability of an `EGPPDecisionNode` or an associated case distinction (`prob_desc`) is decreased from `0.5` to `0.2`, whereas the occurrence probability for a loop construct (`prob_loop`) is increased from `0.0` to `0.02`. Furthermore, in this case, only model instances that have at least one loop construct are included in the evaluation. Due to the changed characteristics of these EGPP instances, only model sizes up to `400` model elements are considered.

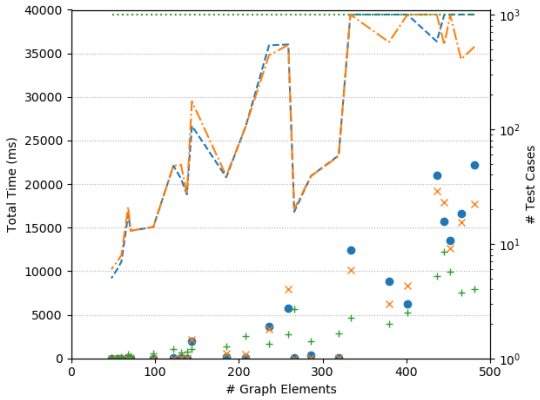


(a) *Low Performance - no Loop*

(b) *High Performance - no Loop*



(c) *Low Performance - Loop*

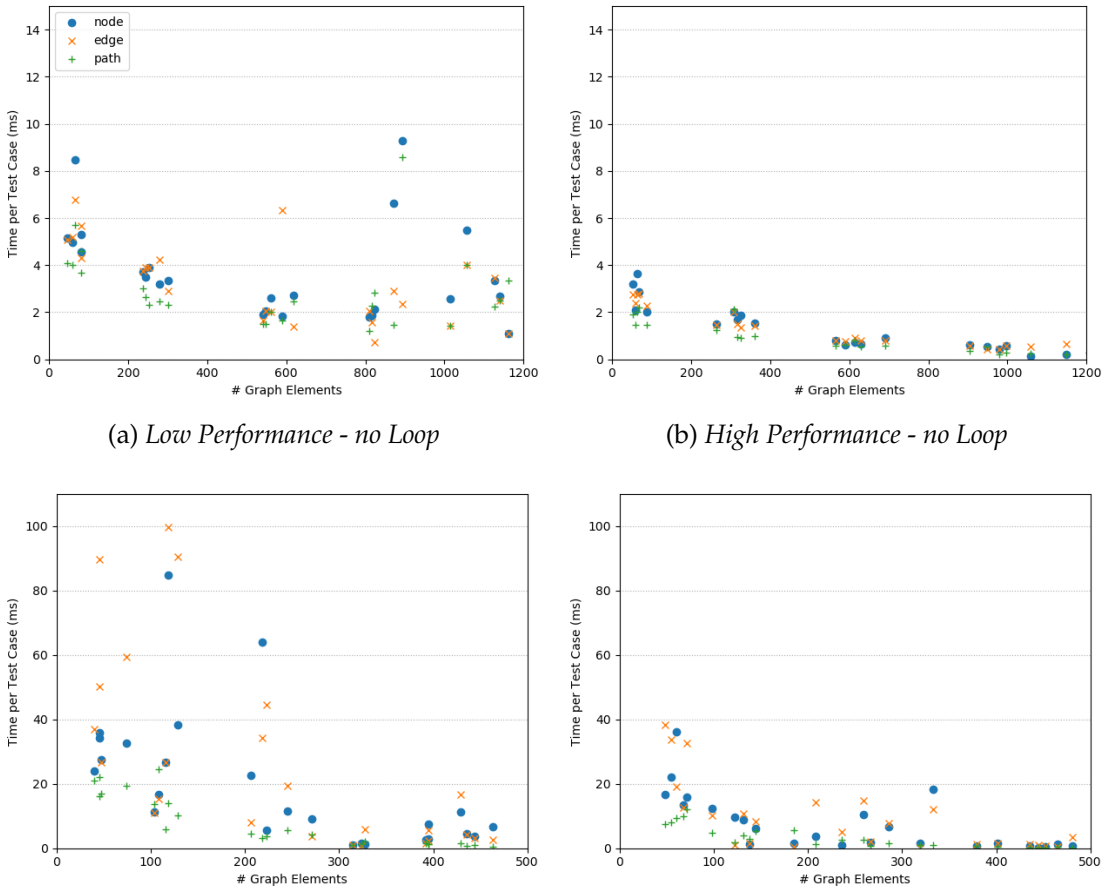


(d) *High Performance - Loop*

Figure 13.4: Illustration of total computation time per Test Case with by means of graph size and the number of Test Cases in relation to graph size

Figure 13.5 is structured in the same way, but in each graph, the average time to generate a test case (`Time per Test Case`) is considered with the `Number of Graph Elements`. The semantics of the visualization of the discrete measured values remains unchanged.

Looking primarily at the scalability of the implementation, it is noticeable in figure 13.4a and figure 13.4b that all considered model instances can be transformed into test cases in under `800ms`, respectively `300ms`. For the instances with loop constructs, the scale was adjusted, wherein this case the generation of test cases does not exceed the `40s`, respectively `23s`. Overall, the time for generating the test cases is in a very low range.



(a) *Low Performance - no Loop*

(b) *High Performance - no Loop*

(c) *Low Performance - Loop*

(d) *High Performance - Loop*

Figure 13.5: Illustration of computation time per Test Case with by means of graph size

Especially against the background of practical test model sizes, which based on empirical values are around 100 model elements, the upper bound of the computation time is reduced again clearly. If the charts in figure 13.5 are utilized for the argumentation on the scalability, it can be seen that the average calculation time per test case decreases as the number of model elements increases. Two effects are responsible for this. On the one hand, the initialization phase of the prototypical implementation is included in the average calculation time per test case, which has a greater effect with less extensive models and fewer test cases. On the other hand, especially in the diagrams 13.5c and 13.5d, a stronger decrease of the average computation time can be determined, which is justified by the technical implementation through data flow analyses. In particular, already traversed parts in the context of loops can be transferred to a result without repeated evaluation, which represents a clear saving. Altogether the concept for the generation of the test cases from EGPP instances has excellent scalability and therefore offers potential for application in an agile model-based development context.

Furthermore, the adaptation mechanism of Test Case Generation, which is based on a subsumption hierarchy of coverage criteria, is evaluated against the background of the

empirical data. In figure 13.4a and figure 13.4b, the measured values for the number of generated test cases show an identical trend for all three criteria considered. Normally, one would expect the curves for *Edge Coverage* and *Path Coverage* to be above the curve for *Node Coverage* in each case. Here, the cause is the implementation of the generator to create the EGPP instances. For example, for branches, all cases are implemented by a combination of `EGPPNodes` and `EGPPTransitions` in each case, even if the case does not contain any instructions. In the context of purely structural coverage measures, this evolves the number of test cases uniformly. In the context of EGPP instances with loop constructs, this phenomenon also applies, but it can be partially seen that the numbers of test cases increase from *Node Coverage* to *Edge Coverage* to *Path Coverage*. Based on this, the use of the aforementioned hierarchy represents a valid solution, because the initial situation concerning the number of test cases is only increased as required, starting from a small quantity, and thus results in the smallest possible quantity of test cases about the utilized metrics.

Finally, we discuss the blacklisting mechanism, which is based on the set of generated test cases just explained. Unfortunately, the empirical data do not provide any information on this, but in the worst case, this filtering of the test case set cannot lead to any further reduction. In the best case, the already small number of test cases is further reduced, which benefits the overall process. Overall, the presented Test Case Management approach shows predominantly positive characteristics within the scope of the evaluation, which again underlines its suitability.

13.3 Model-based Abstract Test Execution

In this section, the Abstract Test Execution approach presented in chapter 10 is evaluated. For this purpose, how the evaluation is carried out is explained first, before the actual evaluation is carried out afterward. In the first place, a qualitative evaluation is carried out, which is then complemented by a quantitative consideration concerning scalability.

Evaluation Setup

The evaluation of the Abstract Test Execution concept is composed of four main steps, which are illustrated in figure 13.6.

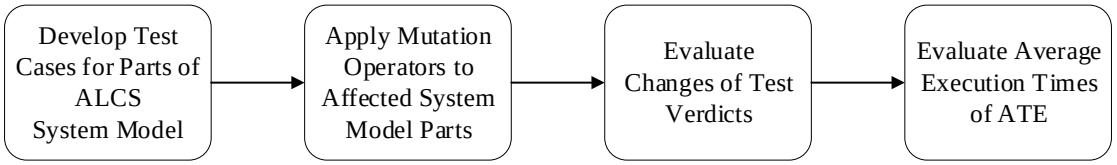


Figure 13.6: Core aspects of evaluation

This evaluation is performed on the ALCS use case, whereas the corresponding Omni Model has already been described in section 12.2. In particular, the explanations of the Omni Model show the relevant System and Test Models used in the evaluation (see `doCtrlCrashFlashing` and `doTheftFlashing`). These two System Model parts are very interesting because all types of model elements are included to evoke all different Test Verdicts. Furthermore, the evaluation results published in our conference paper [83] are taken up and presented again in detail.

In the first step of the evaluation, the test cases are generated from the two Test Models for the system parts `doCtrlCrashFlashing` and `doTheftFlashing` of the ALCS. Based on this set of test cases, the modification of the Test Verdicts and thus the functionality of the presented approach is demonstrated in the evaluation.

In the second step of the evaluation, mutation operators are applied to the linked components of the System Model. Similar to the procedure of Mutation Analysis, possibly triggers changes in the Test Verdicts, which shows that all Test Verdicts are possible and that the correct Test Verdicts are determined.

The resulting test verdicts are derived for this purpose as part of the third step. On the one hand, the test cases are evaluated against all System Models by the prototypical implementation in the A3F. On the other hand, the Test Verdict is derived manually for some test cases using the presented set of rules, which ensures the correct determination of the Test Verdicts on a sample basis.

In the final step of the evaluation, the analyses are evaluated by the prototypical implementations about their runtime, from which in turn conclusions can be drawn regarding the scalability of the implementation.

From a technical point of view, a modified variant of the `egpp_mutation_testing` analysis is needed to perform the presented evaluation setup. This is especially because the implementation for the classical Mutation Analysis only includes test cases whose reference executions are evaluated with a Test Verdict `PASSED (pa)`. This behavior is not desired here, since changes in the Test Verdict are of interest in this context. The modified version of the analysis (`egpp_mutation_testing_mod`) performs the Mutation Analysis for any test case, no matter which Test Verdict is derived during the reference run. The following configuration parameter assignment is chosen to perform the evaluation.

```
analysis egpp_mutation_testing_mod(mutt) {
  systemmodel="data_transformer|systemegpp|DataTransformationResult|getOutputElements";
  integrationmodel="im_scoping|omni|IMScopingResult|getFilteredIMModel";
  testmodels="im_scoping|omni|IMScopingResult|getGeneratedMBTModels";
  testcases="egpp_path_generation|testegpp|EGPPPathGenerationResult|getTestCases";
  executor="dataflow";
  executorconfig="5,5";
  mutopconfig="DELETE_TRANSITION,DUPLICATE_NODE,REPLACE_CONST,REPLACE_COND_OPERATOR";
  mutsamplingconfig="OFF";
  mutstrategyconfig="CLASSIC";
}
```

Listing 13.2: A3F configuration for the evaluation of Abstract Test Execution approach

The analysis steps used to preprocess the model information are not shown here, as this is identical to the A3F configurations of the concept part except for the use case-specific parts.

Evaluation

In the context of performing the evaluation steps for the Abstract Test Execution concept, the Test Models are shown in figure 12.8, and figure 12.9 are used. These Test Models cover common interaction sequences with the respective system components and consider some edge cases of the modeling. The test cases are derived by the prototypical implementation of Test Case Generation using the `node coverage` criterion. Based on the generated test case sets (32 test cases for `doCtrlCrashFlashing` and 60 test cases for `doTheftFlashing`), the resulting Test Verdicts are examined.

With the help of the modified Mutation Analysis implemented in the A3F, especially the changes of the Test Verdicts driven by the applied mutations become apparent. Figure 13.8 shows the results of the analysis, which have already been presented in their basic form in [83].

First and foremost, the figure is divided into four segments, with each segment illustrating the data for a mutation operator related to both application examples. For example, in figure 13.8a, the mutation operator `DELETE_TRANSITION` is considered, with the application example `doCtrlCrashFlashing (Crash)` on the left side and the application example `doTheftFlashing (Theft)` on the right side. In each of these subgraphs, the distribution of the resulting Test Verdicts is plotted on the y axis, with the different mutation targets plotted on the x axis. By using the `CLASSIC` strategy for the modified variant of Mutation Analysis, the model elements plotted on the x axis describe the complete list of mutation targets. Excluded at this point are model elements to which the respective mutation cannot or may not be applied.

Besides, in the bars describing the distribution of Test Verdicts, an unchanged Test Verdict compared to the reference execution is counted in the set `Unchanged`. If the test

execution against the mutant results in an altered Test Verdict, it is assigned to the sets PASSED, PROBABLY PASSED, INCONCLUSIVE or FAIL accordingly.

It can already be seen from the evaluation that the executions against the mutated System Models have effects on the Test Verdicts. Furthermore, it can be observed that the selected mutation operators cause changes of the Test Verdict across both use cases towards all theoretically possible Test Verdicts, except for the Test Verdict INCONCLUSIVE. The reason for this is the derivation rules shown in section 10.2.5 for an INCONCLUSIVE Test Verdict, which cannot be stimulated by the chosen mutation operators.

Furthermore, some mutations on certain model elements do not affect the resulting Test Verdict. For example, figure 13.8d plots the data caused by replacing a condition on the model element in question with the value true. In the context of the doCrtlCrash-Flashing model, only the manipulation of the condition at edge 7 causes a change in the Test Verdict. The two test cases that have changed their Test Verdict, in this case, can be identified in the Test Model. By replacing the condition oc_CentralLockingStatus == 1 with the value true, two more test cases can potentially run successfully in the context of the mutants. According to this scheme, further samples are examined and plausibility checked by hand according to the set of rules for deriving Test Verdicts. No cases are discovered in which the prototypical implementation produced a different result than the manual determination of the Test Verdict.

Concerning the newly introduced Test Verdict PROBABLY PASSED, figure 13.8c is discussed in more detail. In the context of the doTheftFlashing use case, it appears that manipulation by the REPLACE_CONST operator on the model elements C' and 6' does not affect the Test Verdicts. This is due to time-dependent variables, which are given special treatment to the evaluation of the Test Verdicts in our approach (see section 10.2.5). This can essentially be attributed to the fact that temporal considerations are not expedient due to a lack of concrete information at the model-level.

Overall, the analysis of the test executions shows that the approach allows for an evaluation of the test cases at the model-level, which, with the known limitations, corresponds to a test execution at the code-level. However, the correctness of the approach has not been proven, only confirmed by a series of evaluations against the use cases presented. A similar picture emerges in all the case studies discussed, which confirms the applicability of the approach.

In addition to the evaluation regarding the resulting Test Verdicts and the effects of mutations on the System Model, the scalability is examined in more detail by recording the execution times. The Mutation Analysis includes a large amount of Abstract Test Executions, so looking at the total runtime gives a good impression of the time required. Figure 13.7 compares the execution times of the modified Mutation Analysis in both application scenarios.

Furthermore, the measurements are again performed on the two platforms introduced in section 13.2. Here, the mapped values result from runs of the modified Mutation Analysis in the context of the prototypical implementation in A3F. Due to the difference

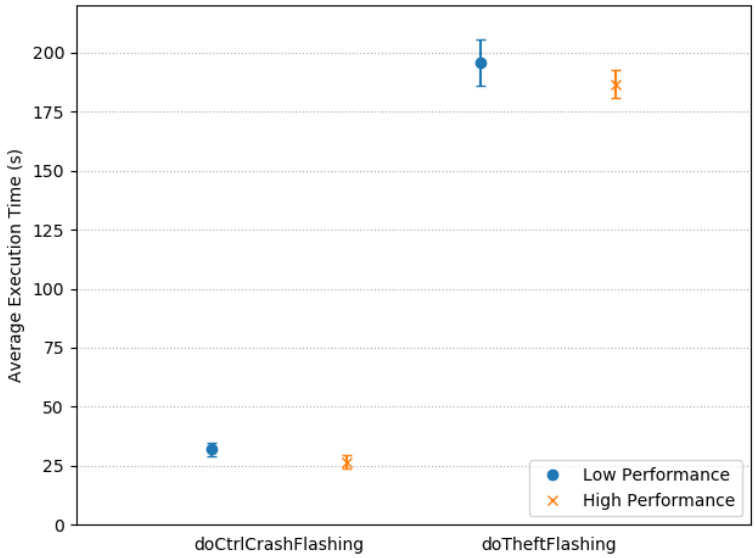
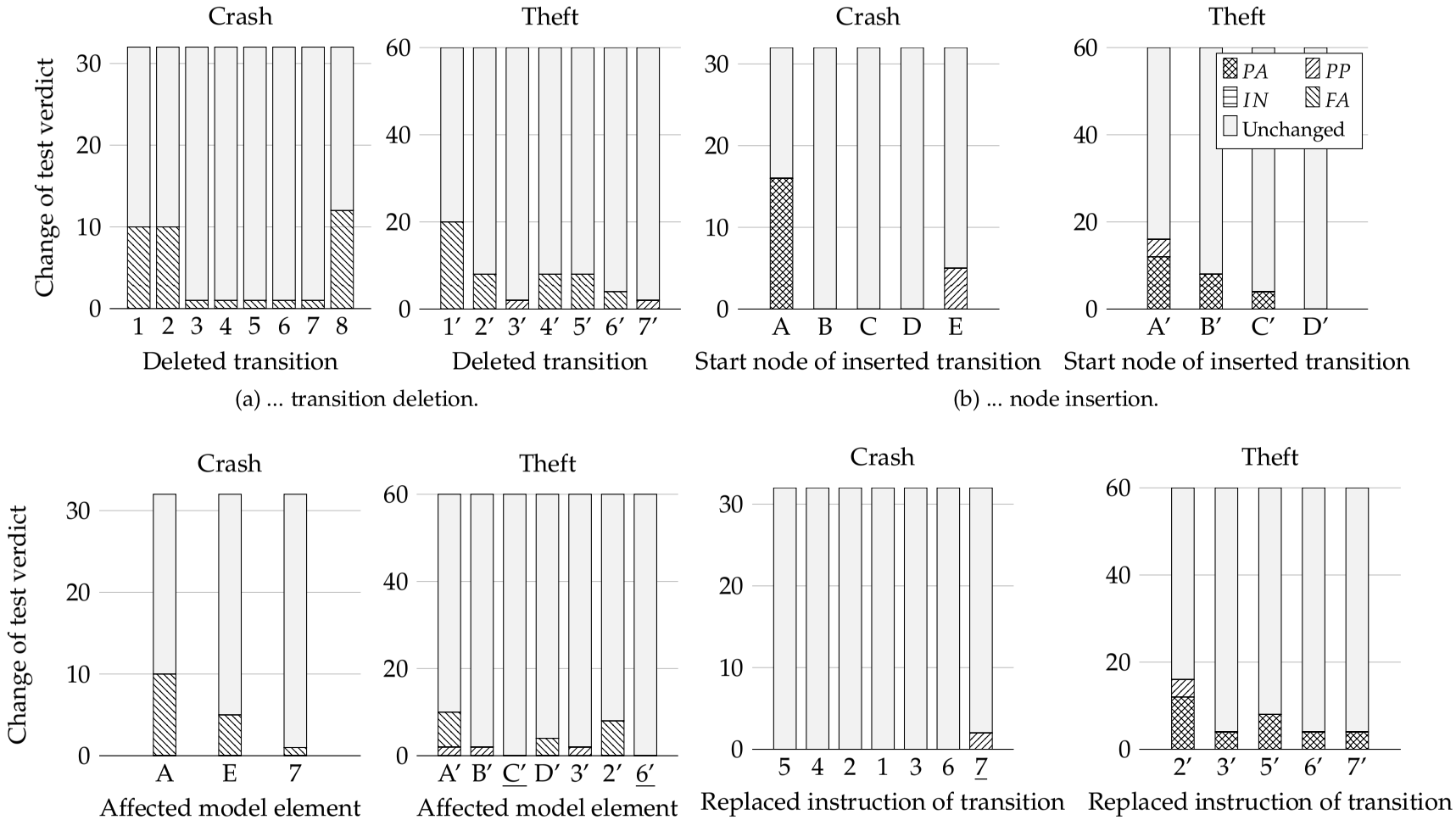


Figure 13.7: Average execution times for Mutation Analysis of `doCtrlCrashFlashing` and `doTheftFlashing` (*Low* vs. *High Performance*)

in the number of test cases, which is even strengthened in the context of the Mutation Analysis, there is a significant difference in the total execution time. For the use case `doCtrlCrashFlashing`, the Mutation Analysis is completed in about 30 seconds on both platforms, whereas in the use case `doTheftFlashing` the total time is about 195 seconds each. Furthermore, in the context of Abstract Test Execution, there is a marginal difference across platforms. Overall, the execution time is still within a reasonable range, i.e. there is potential for improvement concerning the scalability of the approach, however, reasonable applicability is given. Investigations in the context of the other use cases verify these findings, which significantly reduces the probability of a distorted view due to use case-specific phenomena.



(a) ... transition deletion.

(b) ... node insertion.

(c) ... constant replacement.

(d) ... instruction replacement.

Figure 13.8: Results of ... [83]

13.4 Model-based Mutation Analysis

In this section, we evaluate the Model-level Mutation Analysis approach shown in chapter 11. First and foremost, a quantitative evaluation is performed, which compares the TCMAS, as well as the execution time required in different configurations, and compares it with the expectations of experienced testers. Besides, further properties are evaluated in the context of a qualitative discussion.

Evaluation Setup

As part of the evaluation of the Mutation Analysis concept, the steps depicted in figure 13.9 are performed to assess the approach. The experiments are carried out on the models of the case study *Elevator System*, whose Omni Model is presented in section 12.3.

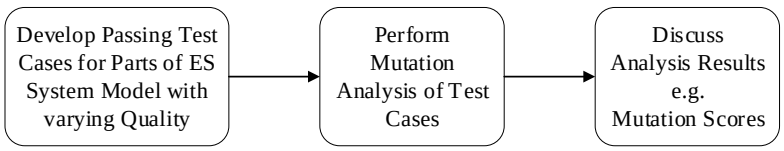


Figure 13.9: Core aspects of evaluation

First and foremost, test cases are to be specified for this purpose, which exhibits a variable quality concerning their level of detail of the verification of a certain subject. Through this set of test cases derived from the Scoped Test Model and shown in figure 12.15, different TCMAS values are expected. Specifically, the test case describing the left path in the Test Model is referred to as `test 1`. Accordingly, the middle path of the Test Model is called `test 2`, and the right path of the Test Model is called `test 3`. In this case, the Test Model is created by a test engineer, whose empirical values are used in the evaluation as a comparison for the classification of the automatically determined metrics.

In the next step of the evaluation, the test cases are subjected to an automated evaluation by the Mutation Analysis approach. Here, the prototypical implementation in the context of A3F is utilized. Listing 13.3 shows an excerpt of the basic configuration of A3F, which contains the parameters of the module implementing the Mutation Analysis.

```
analysis egpp_mutation_testing(mutt) {
  systemmodel="data_transformer|systemegpp|DataTransformationResult|getOutputElements";
  integrationmodel="im_scoping|omni|IMScopingResult|getFilteredIMModel";
  testmodels="im_scoping|omni|IMScopingResult|getGeneratedMBTModels";
  testcases="egpp_path_generation|testegpp|EGPPPathGenerationResult|getTestCases";
  executor="dataflow";
  executorconfig="5,5";
  mutopconfig="REPLACE_CONST,MODIFY_CONST,EXCHANGE_COND_OPERATOR,REPLACE_COND_OPERATOR";
  mutsamplingconfig="OFF";
  mutstrategyconfig="CLASSIC";
}
```

Listing 13.3: A3F configuration for the evaluation of Mutation Analysis approach

To further increase the significance of the experiments, the effect of different optimization mechanisms of the Mutation Analysis concept is considered. For this purpose, a total of three additional configurations are derived from the base configuration (`Config 1`) shown in listing 13.3. `Config 2` modifies the base configuration to use mutant sampling functionality to improve performance. Specifically, at this point, it is set to random sampling using 60% as the sampling rate (`mutsamplingconfig="RANDOM,0.6"`). `Config 3` changes the base configuration in such a way that a different mutation strategy is applied. The previously used `CLASSIC` strategy, which is known to be very computationally intensive, is replaced by the `HIGHERORDER_MIXED` strategy. All other parameters remain unchanged compared to `Config 1`. Finally, `Config 4` introduces another variation of the base configuration, similar to `Config 3`, exchanging the mutation strategy used. In this case, the `HIGHERORDER_CS` strategy is applied, again leaving the remaining parameters unchanged. Across all presented configurations, the same set of mutation operators is used (`mutopconfig="REPLACE_CONST,MODIFY_CONST,EXCHANGE_COND_OPERATOR,REPLACE_COND_OPERATOR"`), which seems reasonable for the presented System Model (see figure 12.14b) from the Test Engineer's point of view.

During the execution of the experiments based on the presented configurations, the TCMASs of the test cases are evaluated on the one hand and the time required for the execution of the analysis on the other hand. The collected information is evaluated and discussed in the last step, which in the end leads to an assessment of the Mutation Analysis concept.

Evaluation

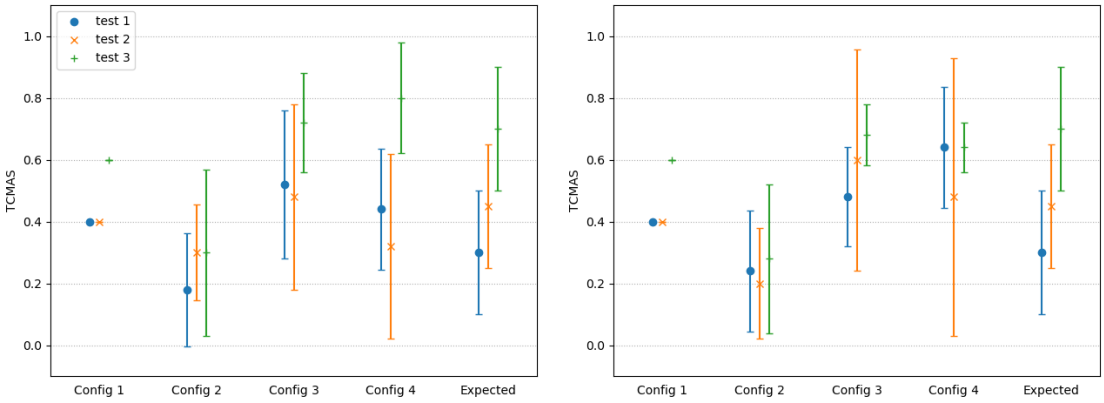
As described in the evaluation setup, the test cases `test 1`, `test 2`, and `test 3` derived from the Test Model are subjected to Mutation Analysis in the context of the different configurations. First, listing 13.4 shows some fragments of a Mutation Analysis Report as provided to the user in the context of A3F.

In the upper part of the report, some statistical values for Mutation Analysis are listed. In the further course, the created mutants are listed first, describing which mutation

operators are applied to which model elements. The conclusion of the general part of the Mutation Analysis is an overview, which shows the different Test Verdicts per test case and additionally quantifies the determined TCMAS.

The lower part of the report lists the individual execution reports of the Abstract Test Execution. These offer almost the same information as already described in section 10.2.6. However, the information regarding the presence of an *equivalent mutant* is given additionally.

Based on such Mutation Analysis Reports, the following evaluations are created. First, the behavior of TCMAS in the different configurations and on different systems is evaluated, which can be seen in figure 13.10.



(a) *Low Performance* (b) *High Performance*

Figure 13.10: TCMAS of test cases 1-3 based on configurations 1-4

The figure is divided into figure 13.10a, which reflects all evaluations for the *Low Performance* platform, and into figure 13.10b, which shows all data for the *High Performance* platform. The two subgraphs each include a quantified assessment of test case quality by the expert on the right side of the plot. The absolute values are negligible since even for an experienced test engineer such an assessment is very difficult. The focus is on the classification of the test cases to each other, which is provided with a relatively high standard deviation.

Considering the data on the evaluation context `Config 1`, we can see that the Mutation Analysis evaluates the two test cases `test 1` and `test 2` identically concerning the TCMAS. Only `test 3` receives a better rating, which is in line with the intuition of the test engineer. However, the range of available values of TCMAS is much smaller than expected, which makes it more difficult for the user to choose a reasonable threshold for a selection of the test case set. One possible reason for such an evaluation is the simplicity of the underlying System Model, which provides relatively few options for applying mutations.

In contrast, the evaluations on `Config 2` provide fewer unambiguous results. On both

platforms, the introduction of random-based mutant set reduction led to a huge scatter in the recorded TCMASs. Also, since the test cases cover only a small portion of the System Model, there is a relatively high probability that all mutations potentially stimulable by the test case are dropped from the set of mutants. This sometimes leads to a TCMAS of `0`. As can be seen in figure 13.11, running Mutation Analysis in `Config 2` leads to a significant reduction in runtime, but the TCMAS can no longer be meaningfully used as a selection criterion.

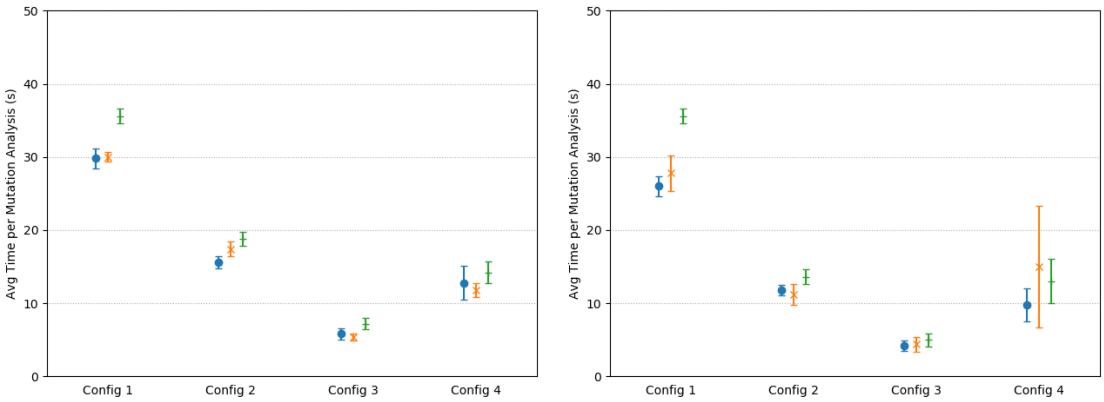
In the context of the evaluations on `Config 3`, the determined values of TCMAS tend to increase, which is due to the mutants. These emerge from a combination of mutations in the context of the `HIGHERORDER_MIXED` strategy. The significant fluctuations in the resulting TCMAS of the test cases `test 1-3` are again due to the random component during the creation of the mutants. Namely, not all mutations are necessarily incorporated into one mutant, but only a certain percentage (see section 11.2.1). The use of TCMAS as a criterion to select a subset of high-quality test cases proves to be problematic in this context.

A similar conclusion can be drawn for the evaluations of the Mutation Analysis in the context of `Config 4`. In each case, the values of TCMAS are distributed over a relatively broad spectrum, and the use of this criterion is again questionable. Furthermore, the advantages of the `HIGHERORDER_CS` strategy cannot be exploited, since the System Model includes only a linear control flow and thus the emergent effects can occur in any case.

In contrast to the ambiguous results in the context of the elevator case study, clearer results are obtained, for example, when performing the Mutation Analysis on the ALCS. We see the reason for this in the simplicity of the elevator System Model, which very much limits the spectrum of mutations and thus, by definition, makes the classification of test cases relatively rough. In particular, the focus of our Mutation Analysis approach is on more complex models, where the evaluation of a test case or a set of test cases cannot easily be done by hand.

In addition to the considerations of the determined TCMASs, figure 13.11 shows the execution times in each configuration.

Overall, the two subgraphs clearly show that the execution times on both platforms are almost identical, which confirms the findings regarding the low influence of the platform from figure 13.7. Moreover, we see that the random-based mutant set reduction (see `config 2`), as well as the mutant set reduction from the combined application of mutations (see `config 3` and `config 4`), produce significant runtime savings. The scatter in execution times that occur in figure 13.11b (`Config 4`, `test 2`) is caused by an outlier that is simply due to a system-side delay in the process. Regarding the evaluation of the execution times, the findings could be confirmed by experiments in the context of the other case studies. The scalability of the Mutation Analysis is largely due to the scalability of the Abstract Test Execution approach. The creation and selection of mutants take up a negligible portion of the total runtime.



(a) *Low Performance* (b) *High Performance*

Figure 13.11: Average execution time for Mutation Analysis of a test case in the respective configuration

Overall, the evaluation shows that the present Mutation Analysis concept is a practical approach, although it must be evaluated concerning the application context and is consequently only fully recommendable in cooperation with an experienced test engineer.

```
1  ================================================================================
2  ==================== Statistics for Mutation Testing Run ====================
3  Test Cases Total: 3
4  Test Cases Mutation Tested: 3
5  Time Elapsed For Run: 82s
6  Average Mutation Score: 0,40
7  ================================================================================
8  Mutants:
9  --------------------------------------------------------------------------------
10 Mutant Name: MUT1337_radcase
11 Mutant ID: MUT1337-2hEmPW2g
12 #Mutations Applied: 1
13 Concrete Mutations:
14   Mutation: EXCHANGE_COND_OPERATOR
15   Applied To: inEtage -> ausserhalb (Etage)
16 --------------------------------------------------------------------------------
17 [... EACH MUTATNT IS SPECIFIED LIKE ABOVE ...]
18 --------------------------------------------------------------------------------
19 ================================================================================
20 Legend: <MUTATIONSCORE>: <REFERENCE_VERDICT> -> (PA,PP,IN,FA) for <TESTCASEID>
21 ================================================================================
22 0,80: PA -> (4,0,0,2) for TC003_MBT-Etage
23
24 [... BEGINNING OF EXECUTION REPORT PER MUTANT AND TEST CASE ...]
25 ================================================================================
26 Mutant Under Test: MUT1337_radcase
27 Test Result: FAIL
28 Mutant Killed: Yes
29 Equivalent Mutant: No
30 Execution Report:
31 --------------------------------------------------------------------------------
32 Test Steps:
33 1. ["_Root__Elevator__Etage0__AnfSchleich == 0 ;"]
34 5. ["_Root__Elevator__Etage0__istInEtage = 1 ;"]
35 6. ["Root__Elevator__Etage0__TasteHolen = 1 ;"]
36 7. ["_Root__Elevator__Etage0__holen == 1 ;"]
37 --------------------------------------------------------------------------------
38 System Steps:
39 0. Initial0 (anforder)
40 2. gen-sub-1-of-warten (warten)
41 3. gen-sub-2-of-warten (warten)
42 4. gen-sub-2-of-warten_SPLIT_0 (warten) (["AnfSchleich = 0 ;"])
43 8. gen-sub-1-of-gedrueckt (gedrueckt)
44 9. gen-sub-2-of-gedrueckt (gedrueckt) (["holen = 1 "])
45 --------------------------------------------------------------------------------
46 Executor Log:
47 ERROR: Guard
48   "Root__Elevator__Etage0__TasteHolen && ( Root__Elevator__Etage0__AnfSchleich)"
49    ignored! Got candidate(s): "[1 && ( 0 )]"!
50
51 WARNING: Startpoint not specified! Using initial node of this (sub)graph.
52
53 INFO: We passed all VPs and last VP was solved with the specified Endpoint!
54 --------------------------------------------------------------------------------
55 [... OTHER EXECUTION REPORTS ...]
```

Listing 13.4: Mutation Analysis Report for an exerpt of the Elevator System

14

Discussion on the overall MCSTLC

Since different evaluations were carried out in advance for the individual process steps of the MCSTLC, the overarching life cycle itself is finally subjected to a qualitative evaluation. For this purpose, different properties are examined, as they were already considered in a similar selection in the context of the evaluation of the Omni Model approach. Specifically, the MCSTLC is evaluated for *Adaptability*, *Extensibility*, *Reusability*, and *Maintainability*.

Adaptability

In terms of the application context, the MCSTLC offers some possibilities for adaptation, which are attributable to the concepts of the substeps. Due to the underlying Omni Model concept, the MCSTLC can be built on top of almost all model-based development contexts. The ability to adapt the approach is due to the decoupling of the processing steps from the original ways of representing the model information. This continues in the incorporation of meta-information, where full flexibility is given to the expression of the Aspects as long as it is deterministically quantifiable information.

The adaptability in the context of the MCSTLC exceeds that except for the creation and maintenance of the additional models and meta information, no change regarding the concrete MCSTLC instance is necessary. Even in the context of a hybrid use of code-based and model-based techniques during development, an application of the MCSTLC is possible without further restrictions. Considering all the concepts of MCSTLC against the background of possible application contexts, the approach shows a very good overall adaptability.

Extensibility

Concerning the *Extensibility* of the MCSTLC, similar arguments can be made as for *Adaptability*. For example, the adaptability of the set of Aspects includes the extensibility, whereby theoretically any number of such metainformation can be added and used for processing.

However, the MCSTLC itself can be extended. Thus, an extension of the set of process steps by additional use case-specific processing steps is conceivable. An example of this is an enhancement of the test quality to other criteria that are described by an application domain-specific standard, which requires an analysis of the test case set in a further process step. Such a process step can be integrated without any problems as long as the exchanged data structures are only extended and not modified. Due to the modular structure of the MCSTLC, the concepts can be extended and combined in other ways as desired. As part of the prototypical implementation, there are analysis modules deriving processing artifacts for other tooling and bringing its results back into the processing chain. This demonstrates that there are few limits to extensibility.

Reusability

The MCSTLC adapted or extended for the application context allows smooth reusability in other use cases. For this purpose, only the components concerning the adaptation to the original modeling approaches need to be exchanged or adapted. As long as the modeling languages and metamodels remain identical, even these adaptations are not necessary and the MCSTLC can be reused without further action.

Looking at the components of the MCSTLC, the application of the Omni Model concept enhances reusability for the development artifacts as well. This linking of concepts across modeling domains promotes synchronization between development domains, which is comparable to continuous reviews. This implicitly improves quality and enables reusability in the sense of a complete package of models in other contexts.

Maintainability

Finally, the MCSTLC is considered and discussed concerning its maintainability. For this purpose, maintainability can be considered about the model artifacts that are processed within the framework of the MCSTLC. The configuration parameters of the individual process steps have to be maintained as well so that the automated processing can be performed in a meaningful way. Regarding the model artifacts, the maintainability is not affected by the application within the MCSTLC. For example, if the maintenance of the system modeling in the context of the company's modeling language or approach was problematic before, the application in the context of the MCSTLC does not deteriorate it. The opposite is true: through the aforementioned synchronization of the modeling domains in conjunction with suitable tool support, maintenance can even be improved. The only exception is the additional Integration Model, whose maintainability without appropriate tool support represents a sensitive point.

From the point of view of the configuration parameters of the MCSTLC, maintainability is given. The fact that all parameters are stored centrally in a configuration means that there is no danger of parts being overlooked and possibly not maintained. Furthermore, the absolute number of parameters that can be manipulated by the user is

limited, so that complexity remains on a low level. Furthermore, in the context of the prototypical implementation, it is possible to fall back on default parameters in many places, which do not usually represent the optimal configuration but do further simplify maintainability and still produce useful results.

Considering all evaluated properties of MCSTLC together, the positive properties of the approach outweigh the shortcomings, whereas the identified issues can be remedied in the context of future work.

Part V

CONCLUSIONS AND OUTLOOK

15

Conclusions

In the context of this work, a concept including a prototypical implementation has been developed which, by adapting and combining different modeling and testing techniques, created a solution to tackle the complexity of testing and to evaluate test cases against a system in the very early phases of MDSD. In particular, this offers the possibility of detecting errors introduced into the system in early phases of development at a correspondingly early stage and thus minimizing their impact throughout the entire product life cycle. In response to the research question posed at the beginning

How to carry out a testing life cycle on the model-level to counteract complexity and severe defects, improving the overall quality of development artifacts in MDSD scenarios?

the following answer can be given.

The basis of any software testing life cycle is first of all the knowledge about the specified system to be developed and all influencing factors. From the point of view of a model-centric approach to the problem, this knowledge base was created by first developing a concept that enables a lightweight integration of model information from all modeling domains (influencing factors). This so-called Omni Model, which preserves the Separation of Concerns to the modeling domains, comprises a flexible set of models and links them employing an Integration Model. Furthermore, this Integrated Model Basis allows to integrate meta information into the model landscape and to benefit in subsequent processing steps. In this context, Research Question 2 as well as Objective 2 are covered by the presented Omni Model concept, which has already been discussed throughout the conclusions in section 7.5.

Based on the Omni Model, a Model-Centric Software Testing Life Cycle (MCSTLC) was implemented that reinterprets the phases of a traditional software testing life cycle at the model-level and explicitly provides mechanisms for determining and improving the quality of test cases. Primarily, however, this widely automated MCSTLC supports the developer and tester in their model-based development work. The first process step of MCSTLC, the so-called Test Case Management, enables the user to reduce the comprehensive Test Model to the current test focus by specifying constraints on the Omni Model information and subsequently evaluating them. In interaction with other process steps of the MCSTLC the reduction, prioritization, and selection of test case sets are realized. This process step represents the response to Research Question 3, whereby

all the objectives identified at the beginning (see Objective 3) could be implemented. As a closing remark in chapter 8, a more detailed conclusion has already been presented in section 8.5.

The next process step of the MCSTLC, the so-called Test Case Generation, realizes the derivation of a test case set from the reduced Test Models, which were determined in the previous step under consideration of external regulations, such as coverage according to explicit metrics. Furthermore, this approach considers the quality of the generated test cases by filtering according to a Mutation Analysis based metric. This concept provides a promising solution to Research Question 4 and the resulting objectives (see Objective 4). A detailed conclusion can be found in section 9.5.

The following process step of the MCSTLC, the so-called Abstract Test Execution, enables the execution of the test case set at the model-level. Here, the need for a transition to code artifacts was explicitly avoided, as this would nullify the benefits of early applicability. The implementation, therefore, uses an internal model representation of the original models, to accomplish on the one hand through search algorithms and on the other hand through data flow analysis techniques the evaluation of the available test cases against the System Model. The gained knowledge is made available to the developer and tester to achieve an improvement in subsequent iterations. Based on Research Question 5, any identified objectives were fulfilled by the concept presented in the thesis (see Objective 5). Regarding this part of the thesis, a concluding discussion has been performed in section 10.5.

In addition to the pure execution of the test cases, the Abstract Test Execution methodology is applied in the last process step of the MCSTLC. The so-called Mutation Analysis on model-level is primarily concerned with the evaluation of the generated test case set for its quality regarding typical error types. This process step, which is particularly related to the already mentioned test case generation, represents an excursion in the classical life cycle and can therefore be optionally integrated. In this context, Research Question 6 was determined at the beginning of this thesis, whereby the identified objectives subsumed under Objective 6 are all covered by the presented Omni Model concept, which has already been discussed in the conclusions of section 11.5.

The evaluation of the developed functionalities has confirmed their usefulness and applicability with small weaknesses in the context of the prototypical implementation. In particular, this affects the scalability of the current, not optimized, implementation of Abstract Test Execution, used in the context of Mutation Analysis. Likewise, the Maintainability of the Integration Model information, as part of the Omni Model, was identified as a potential source of error for the usability of the generated results, which, however, can essentially be remedied by suitable tool support. However, the positive aspects and opportunities created by the novel MCSTLC outweigh the negative ones.

16

Outlook

Finally, an outlook on further work and future fields of application of the presented concept is given. In the concluding sections of the concept chapters, we have already provided an outlook on the further development of the respective concept. Based on the findings of the evaluation, there are some further aspects that, in our view, represent a logical continuation of the presented approaches. In the context of the *Omni Model*, we see the need for a well-founded concept that supports the creation and maintenance of the Integration Model on the process side and provides the basis for suitable tool support. This applies to the creation of model transformations to the internally used model representation. This is expected to lead to a significant improvement in terms of the resilience of the model information, which increases confidence in the results of the MCSTLC and, ultimately, user acceptance of the solution. Furthermore, we see another point in the development of the Integration Metamodel in terms of the incorporation of code artifacts. Since code represents a kind of modeling, and thus the transition and applicability of the concept would merge from the model-level to the code-level, we see high potential in the expansion/refinement of these concepts. Such an extension has already been envisaged in the Integration Metamodel, but has not been pursued or evaluated further due to the scope of the work.

In the context of the *Test Case Management* concept, the optimization of the Aspect-based specification of the test focus represents a possible starting point for further work. Here, an automated derivation of aspect constraints can represent possible tool-side support for the user, whereby the user's knowledge of the information in the Integration Model no longer needs to be detailed. From a conceptual point of view, however, this area of the MCSTLC represents a unit, i.e. few further topics are suggested here.

The process step of *Test Case Generation* offers a variety of further topics. For example, it could be investigated to what extent procedures for deriving test cases from models that do not fall into the category of coverage criteria can complement the existing coverage measure-driven criteria. At this point, we believe there is a very large potential to make the resulting set of test cases even more effective and targeted. The use of learning algorithmic approaches is conceivable, whereby empirical data on effective test cases can be applied to the current use case. However, this research direction should be treated with caution, especially in the testing context, since context plays a crucial role in testing, which is reflected in the selection of learned parameters. Similarly, the use of mutation in terms of exploratory expansion of the test case set would be a conceivable technology to extend the existing concept. Another aspect of Test Case

Generation, which should be investigated in further work, is the diversification of the test quality criterion, which is currently used to filter the test case set (see TCMAS). Different characteristics apart from the Mutation Score, which evaluates the ability of a test case/test suite for frequently occurring fault types could determine an even more objective picture of a high-quality test suite.

The process step *Abstract Test Execution* probably offers the most potential for further work. Similarly, as the Mutation Analysis was developed to a practicable solution by a multitude of optimizations, the algorithms for test execution can be optimized in many areas. This can be accomplished either by parallelization variants or suitable preprocessing of model data. An alternative development of the current approach, which evaluates all possible paths in the system, would be a user-guided evaluation of the paths. This could significantly limit the path space to be considered since irrelevant parts of the execution can be identified and skipped by the user at runtime (similar to state-of-the-art debugging in an IDE). However, the application context of such a solution would no longer be as automatable as the current concept. However, it would open up new application areas, such as model-based debugging in the early stages of development, which is a more interactive version of the current usage.

For the process step of *Mutation Analysis*, further work concerns the set of Mutation Operators. In particular, the possibility to define Mutation Operators in a context-specific way would allow a more effective application of the Mutation Analysis concept. In its current form, the set of Mutation Operators represents a cross-section of error types based on research and experience, which have been generalized and implemented in the context of our internal model representation. While this in principle allows for flexible use of the Mutation Analysis methodology, it is not necessarily as specific as it would need to be to achieve better results. In the sense of effective use of the Mutation Analysis concept a learning algorithm approach would be conceivable, which offers decision support for the configuration of this analysis for the respective use case. This can also be accomplished by a complementary set of modeling guidelines for test modeling, thus addressing the challenge at its origin.

From a process point of view, questions concerning the integration of the MCSTLC methodology into a wide variety of development processes represent a possible starting point. For example, the question arises as to what extent the presented approach can support a form of agile software development in a model-centric approach. Likewise, investigations regarding the possibility of a Model-Level Test Driven Development (MTDD) through our MCSTLC represent interesting aspects of further research topics.

Part VI

Annex

Supplementary Material

CSM System Model Diagrams

This section shows the submodels of the Ceiling Speed Monitor (CSM), which was mapped by the University Bremen in a SYSML model. [38] Starting at the highest level of the system model, the encapsulated SUD is specified with its ports to the environment as well as the data structures communicating via the respective ports are specified. Moreover, the structural breakdown of the SUD blocks into their sub-blocks is defined. It should be noted that the two blocks *TSM* and *RSM* are not included in the original SYSML model of the University of Bremen, but were added for demonstration purposes for later processing steps. The behavior of the CSM block is described in the form of a hierarchical statemachine, which at this level only distinguishes between a CSM that is either *active* or *inactive*. Besides, the behavior in the case of an *active* CSM is detailed, which is realized as a sub-statemachine of the `CSM_ON` state.

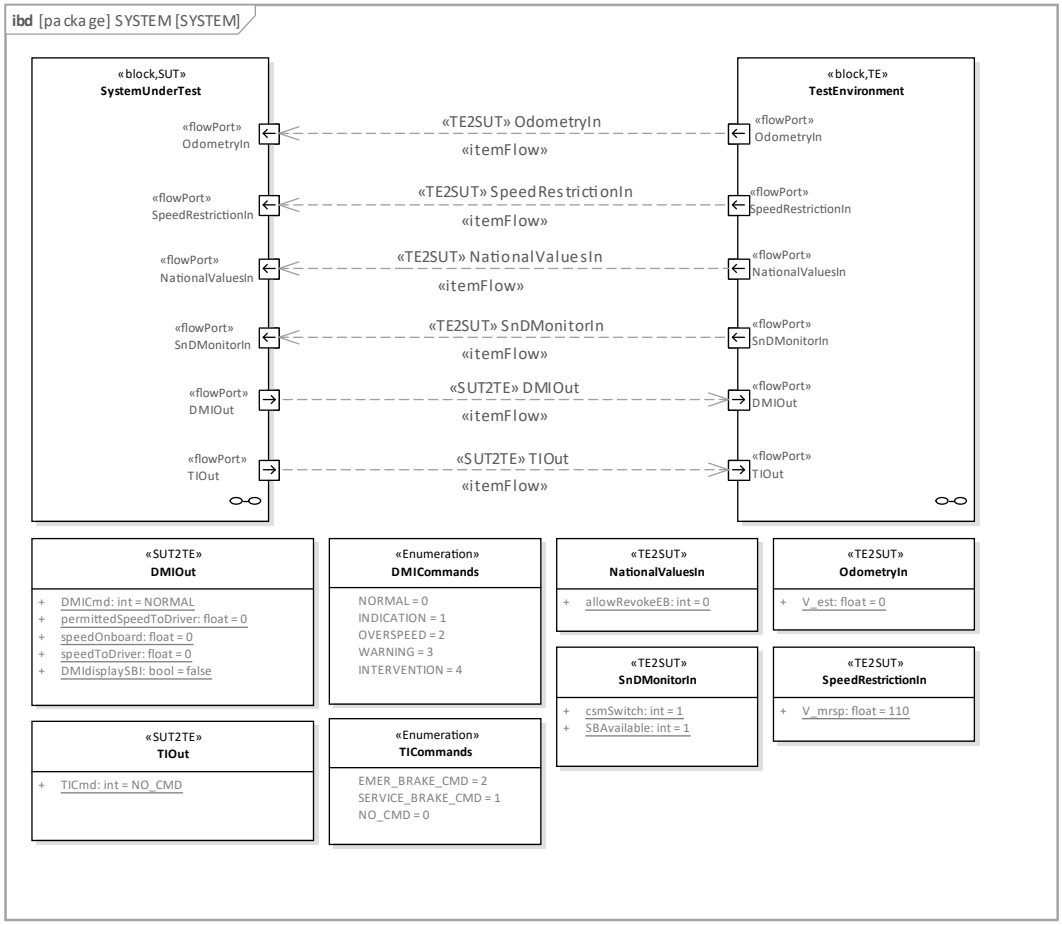


Figure 16.1: Overall view of ETCS Ceiling Speed Monitor model

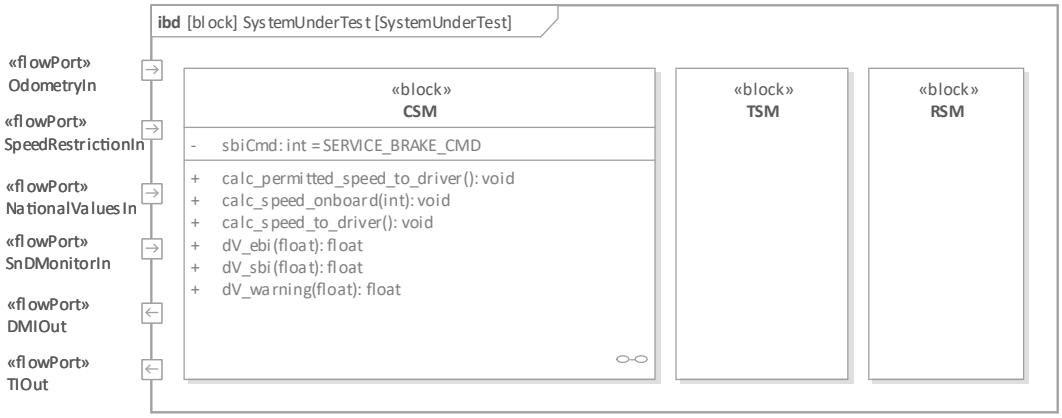


Figure 16.2: SYSML block diagram of ETCS Ceiling Speed Monitor model

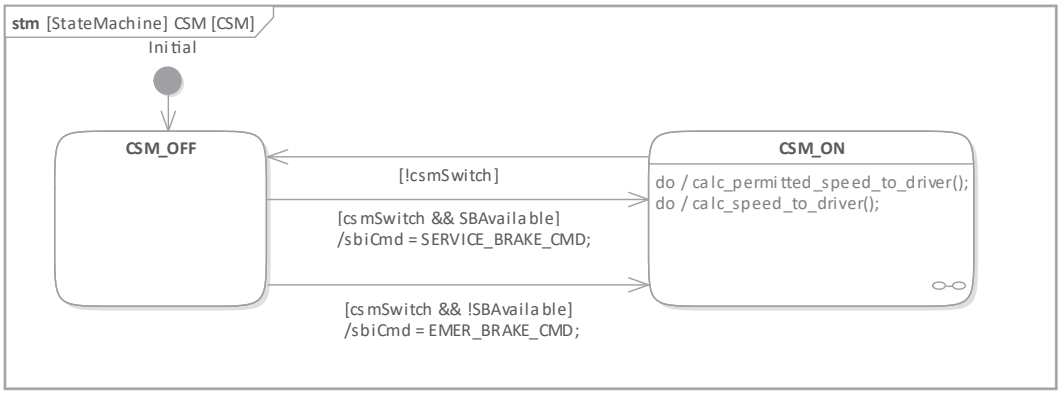


Figure 16.3: SYSML statemachine diagram of ETCS Ceiling Speed Monitor model

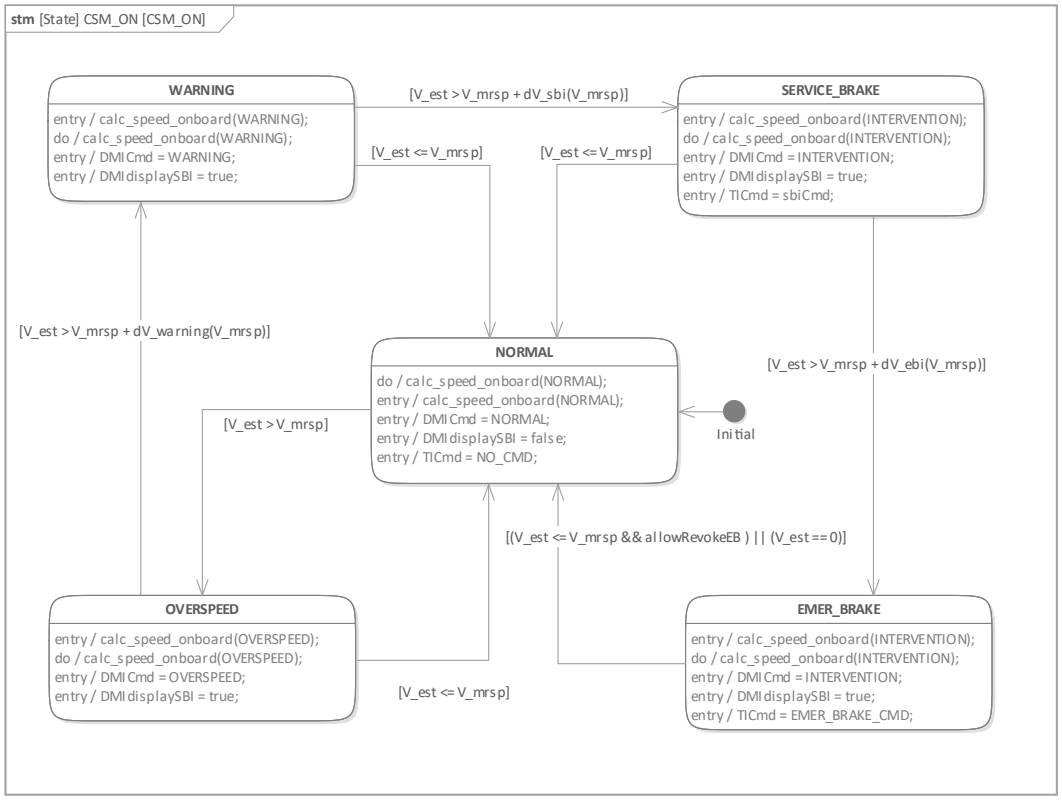


Figure 16.4: SYSML sub-statemachine diagram of ETCS Ceiling Speed Monitor model

Enterprise Architect Metamodel Information

In this section, the metamodel of the Enterprise Architect modeling tool is discussed in more detail. The metamodel essentially reflects the aspects of the Enterprise Architect data storage layer.

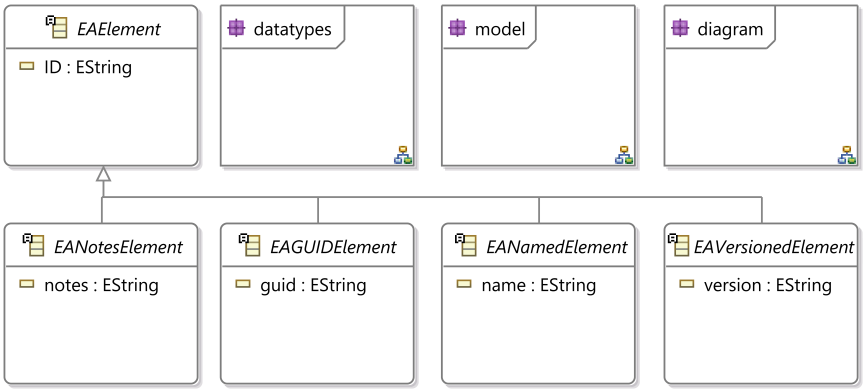


Figure 16.5: EA metamodel

Table 16.1: Metamodel element descriptions for figure 16.5

Concept	Description
`EAElement`	The most abstract concept of the enterprise architect metamodel defining a unique identifier per element
`EANotesElement`	Concept for an element specifying additional information as a note
`EAGUIDElement`	Concept for an element with a globally unique identifier
`EANamedElement`	Concept for an element specifying a name attribute
`EAVersionedElement`	Concept for a versioned element

Further, the metamodel structures its model elements into three sub-packages, namely `datatypes`, `model`, and `diagram`. First, the `datatypes` package is examined more closely.

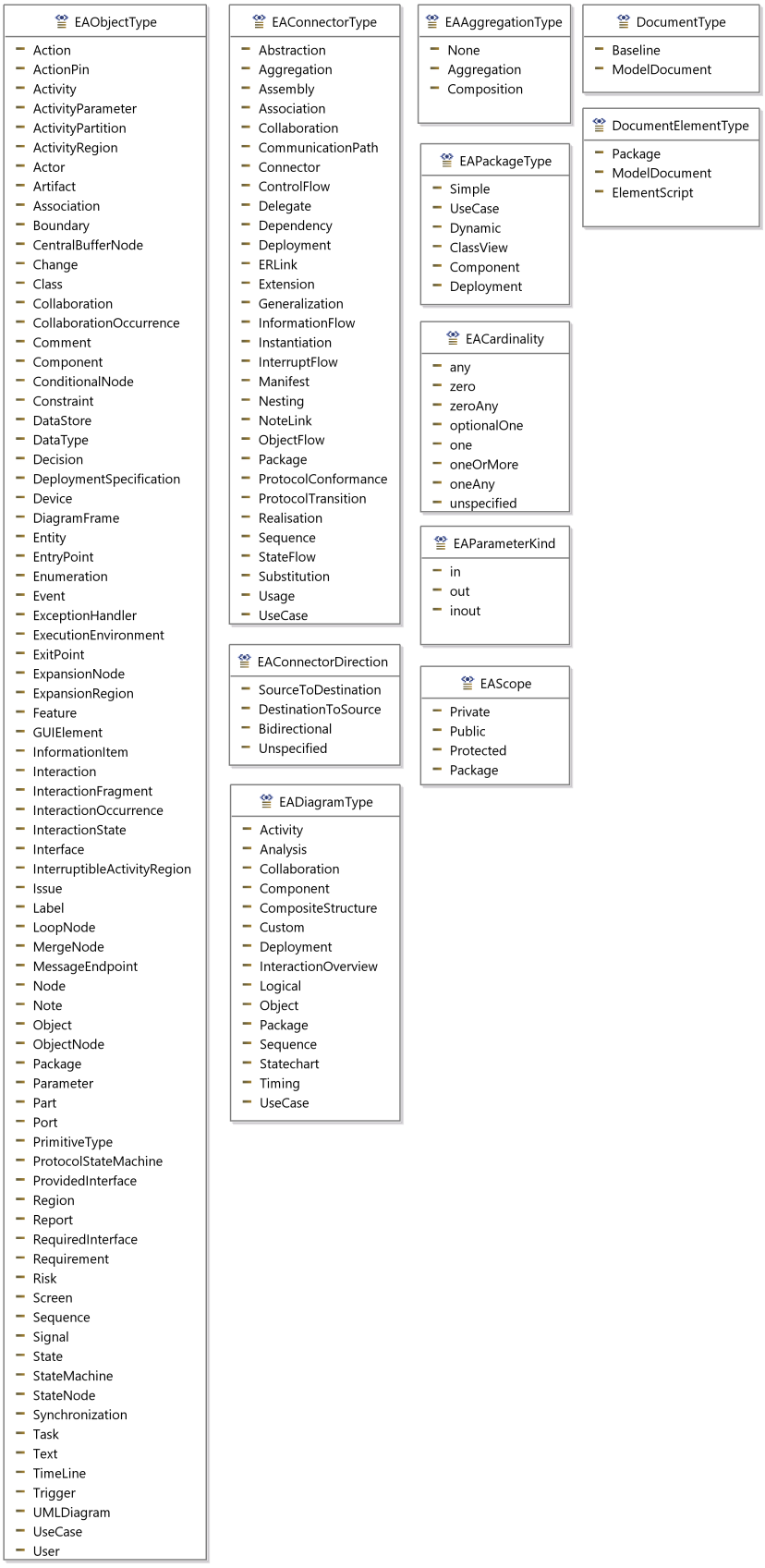


Figure 16.6: `Datatypes` package of the EA metamodel

Table 16.2: Metamodel element descriptions for figure 16.6

Concept	Description
`EAObjectType`	Enumeration concept for the supported object types
`EAConnectorType`	Enumeration concept for the supported connector types
`EAConnectorDirection`	Enumeration concept for the supported connector directions
`EADiagramType`	Enumeration concept for the supported diagram types
`EAAggregationType`	Enumeration concept for the supported special types of a aggregation connector
`EAPackageType`	Enumeration concept for the supported package types
`EACardinality`	Enumeration concept for the specification of connector's cardinality
`EAParameterKind`	Enumeration concept for the supported kinds of a parameter
`EAScope`	Enumeration concept for the supported scopes of elements
`DocumentType`	Enumeration concept for the specification of the document type
`DocumentElementType`	Enumeration concept for the specification of different element types

The `model` package is examined in more detail below.

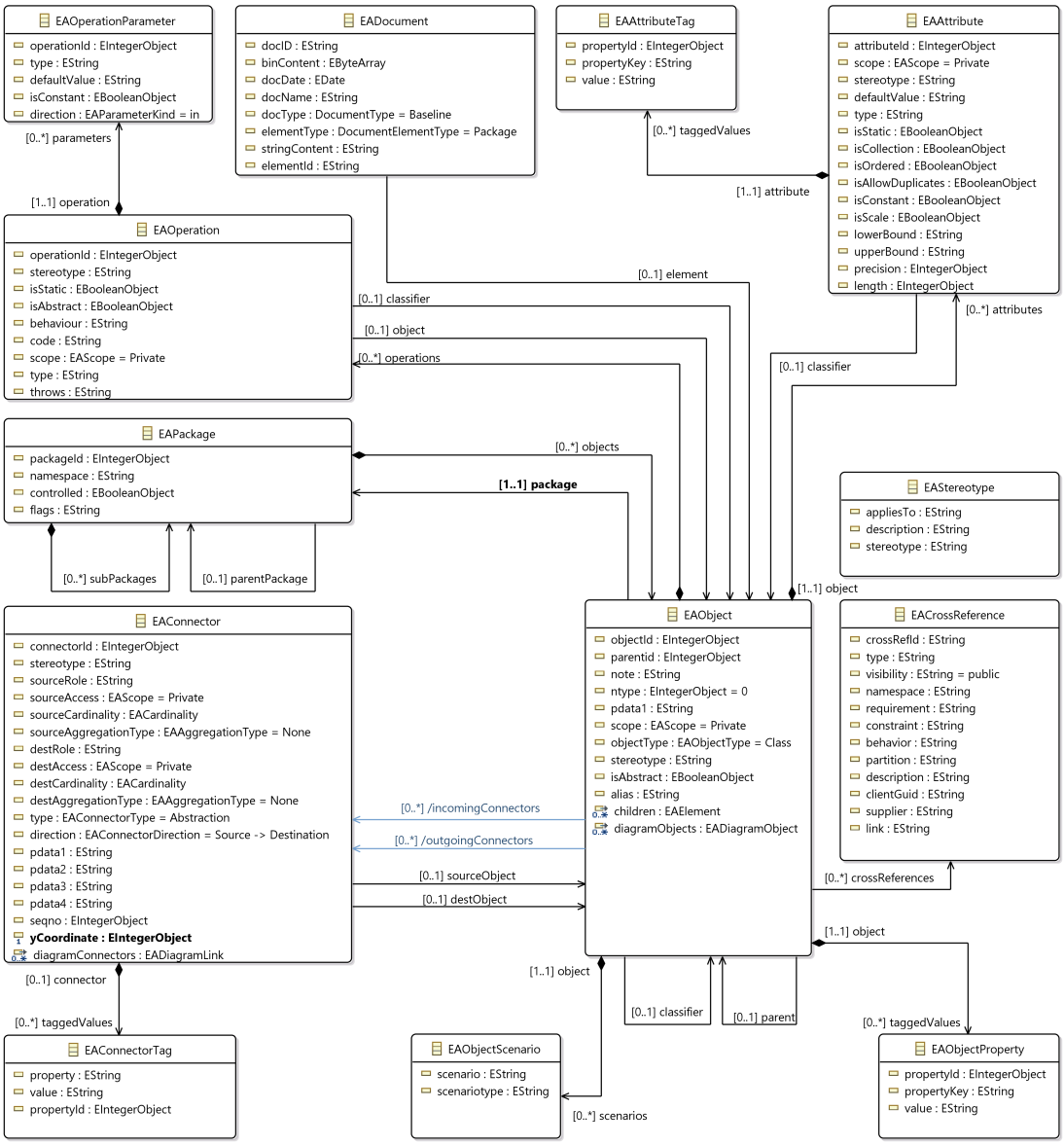


Figure 16.7: `Model` package of the EA metamodel

Table 16.3: Metamodel element descriptions for figure 16.7

Concept	Description
`EAOperationParameter`	Concept representing a parameter of a `EAOperation`
`EADocument`	Concept capturing a document file
`EAAttributeTag`	Concept for additional tag information of a `EAAttribute`
`EAAttribute`	Concept for attribute of a `EAObject`
`EAOperation`	Concept representing an operation of a `EAObject`

Table 16.3: Metamodel element descriptions for figure 16.7 (continued)

EAPackage	Concept for a package structuring sets of objects
EAStereotype	Concept for specifying a stereotype of an element
EAConnector	Concept for a connector between EAObjects
EAObject	Concept for an object connected to various other detailed concepts
EACrossReference	Concept capturing custom references to other concepts of the metamodel
EAConnectorTag	Concept for additional tag information of a EAConnector
EAObjectScenario	Concept putting a EAObject in a certain scenario context
EAObjectProperty	Concept capturing additional property information of a EAObject

Finally, the diagram package is discussed in more detail.

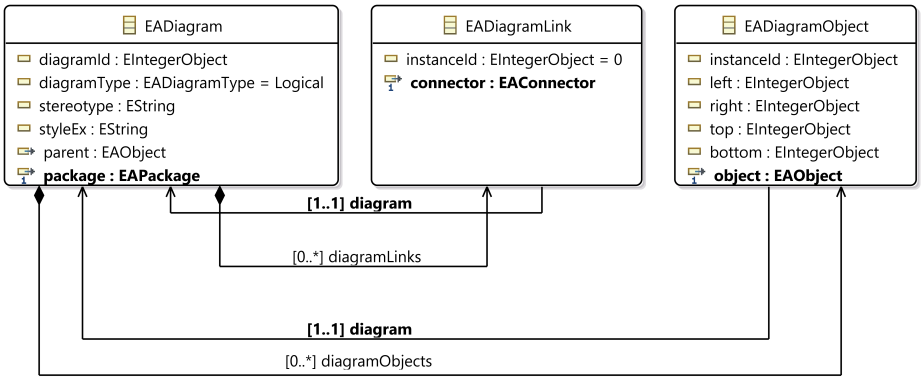


Figure 16.8: Diagram package of the EA metamodel

Table 16.4: Metamodel element descriptions for figure 16.8

Concept	Description
EADiagram	Concept representing a diagram made of multiple EADiagramObjects and EADiagramLinks
EADiagramLink	Concept for the visual aspects of a EADiagramLink
EADiagramObject	Concept for the visual aspects of a EAObject

UML Metamodel Information

This section deals with a part of the UML metamodel, which shows that GPMLs can also participate in the context of the Integrated Model Basis. The excerpt is based on the model scope of the ALCS modeled in the UML, which serves as a case study in the context of our work. In order to map the system model of the case study in our metamodel, the following concepts have been included in the metamodel (see figure 16.9) Further details are given in the context of table 16.5.

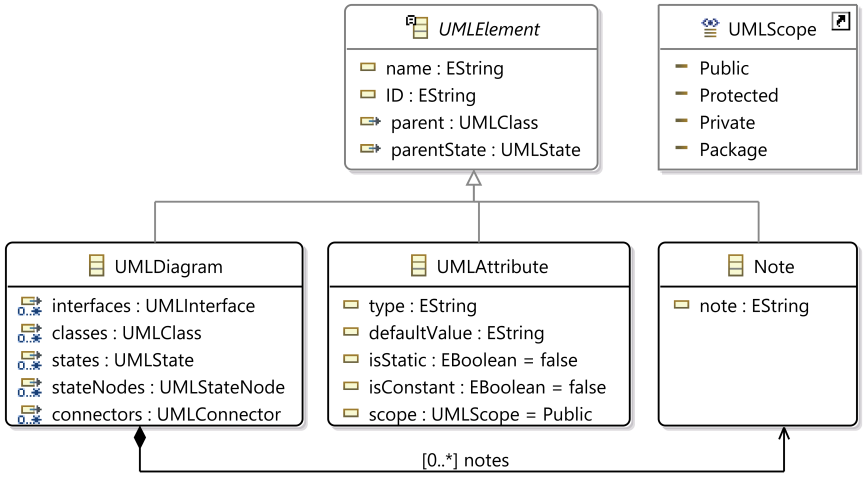


Figure 16.9: Simplified UML metamodel

Table 16.5: Metamodel element descriptions for figure 16.9

Concept	Description
`UMLElement`	The most abstract concept of this metamodel defining among others a unique identifier per element
`UMLScope`	Enumeration concept specifying the scope of a `UMLAttribute`
`UMLDiagram`	Concept for a diagram capturing other model elements
`UMLAttribute`	Concept for the specification of attributes of a `UMLClassElement`
`Note`	Concept for optional information which is captured in a `UMLDiagram`

Apart from the generic constructs, some concepts have been defined in their own packages, which are shown integrated in figure 16.10. Explanations for the elements are again given in table 16.6.

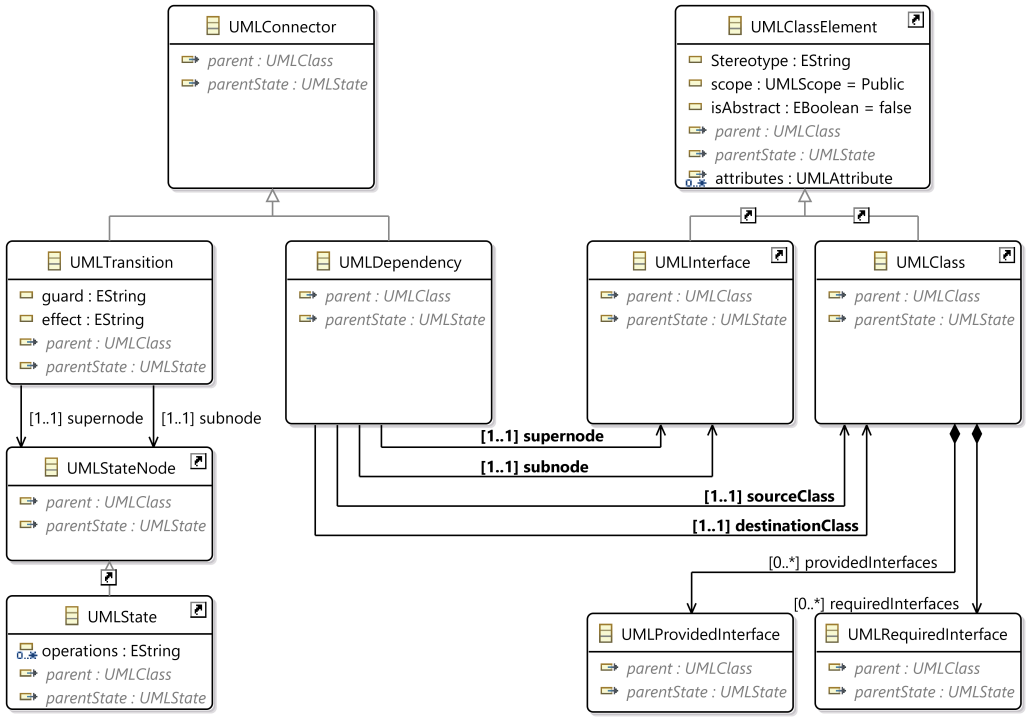


Figure 16.10: Remaining packages of the simplified UML metamodel

Table 16.6: Metamodel element descriptions for figure 16.10

Concept	Description
`UMLConnector`	Abstract concept for a connector between metamodel elements
`UMLClassElement`	Abstract concept for structural elements of the UML metamodel
`UMLTranstion`	Concept for specifying a connection between `UMLStateNodes`
`UMLDependency`	Concept for the specification of hierarchies of model elements
`UMLInterface`	Concept for the specification of a interface
`UMLStateNode`	Abstract concept for states of a statemachine
`UMLState`	Concept for a state which contains additional information about operations
`UMLProvidedInterface`	Concept for a special type of `UMLInterface`
`UMLRequiredInterface`	Concept for a special type of `UMLInterface`

RadCase Metamodel Information

In this section the metamodel of the commercial system modeling tool *radCase* is discussed. This tool is used in the context of embedded systems, which is also reflected in the selection of model elements. The modeling scope and its relationships are shown in figure 16.11, with table 16.7 providing some explanations.

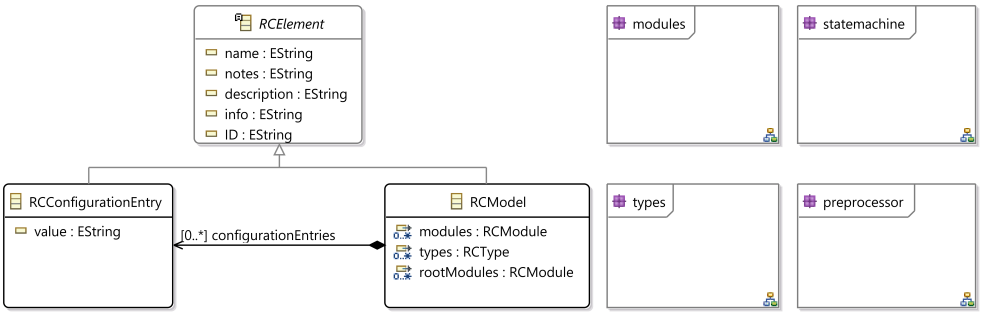


Figure 16.11: Simplified RC metamodel

Table 16.7: Metamodel element descriptions for figure 16.11

Concept	Description
`RCElement`	The most abstract concept of the metamodel specifying among others specifies an identifier
`RCConfigurationEntry`	Concept making up a runtime configuration of the system under development
`RCModel`	Concept representing a container element for a radCase project

The metamodel structures its model elements into four subpackages, namely `modules`, `statemachine`, `types`, and `preprocessor`. First, the *modules* package is examined more closely.

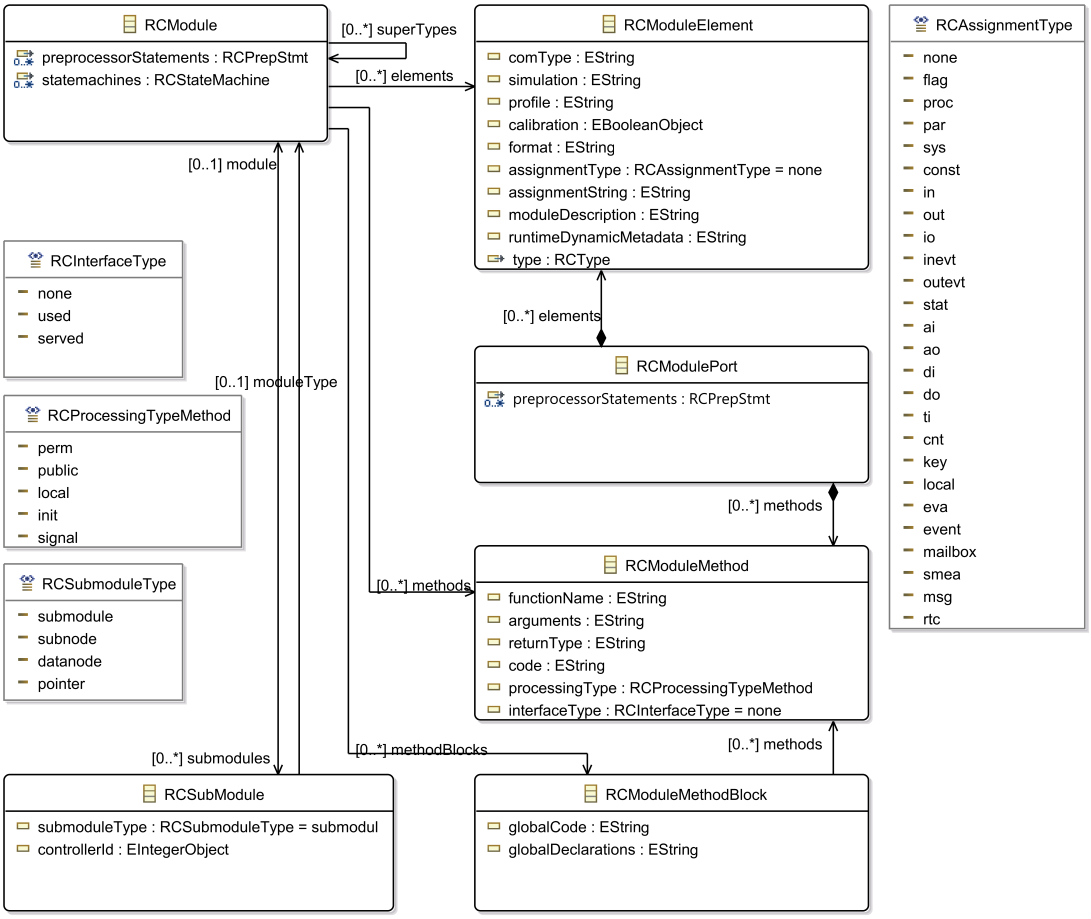


Figure 16.12: `Modules` package of the simplified RC metamodel

Table 16.8: Metamodel element descriptions for figure 16.12

Concept	Description
`RCModule`	Concept for a functional unit of the radCase project
`RCModuleElement`	Concept for the elements of a `RCModule`
`RCAssignmentType`	Enumeration concept for specifying the assignment type of a `RCModuleElement`
`RCInterfaceType`	Enumeration concept for the type of an interface
`RCModulePort`	Concept representing a port of a `RCModule`
`RCProcessingTypeMethod`	Enumeration concept for the specification of a processing type for a `RCModuleMethod`
`RCModuleMethod`	Concept for a method within a `RCModule`
`RCSubmoduleType`	Concept for specifying the type of a `RCSubModule`

Table 16.8: Metamodel element descriptions for figure 16.12 (continued)

`RCSubModule`	Concept enabling the specification of hierarchical module structures
`RCModuleMethodBlock`	Concept encapsulating Code fragments and a set of `RCModuleMethods`

The `statemachine` package is examined in more detail below.

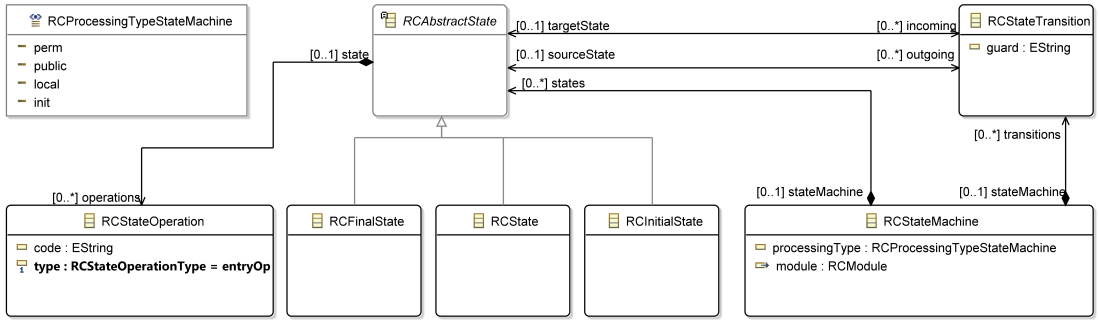


Figure 16.13: `Statemachine` package of the simplified RC metamodel

Table 16.9: Metamodel element descriptions for figure 16.13

Concept	**Description**
`RCProcessingTypeStateMachine`	Concept for the specification of a processing type for a statemachine
`RCAbstractState`	Abstract concept subsuming multiple types of states for a statemachine
`RCStateTransition`	Concept for a transition between states of a statemachine
`RCStateOperation`	Concept for an operation attached to a `RCAbstractState`
`RCFinalState`	Concept for a special kind of state
`RCState`	Concept for a special kind of state
`RCInitialState`	Concept for a special kind of state
`RCStateMachine`	Concept including all the elements of a statemachine like `RCAbstractstates` and `RCStateTransitions`

Next, the `types` package is discussed in more detail.

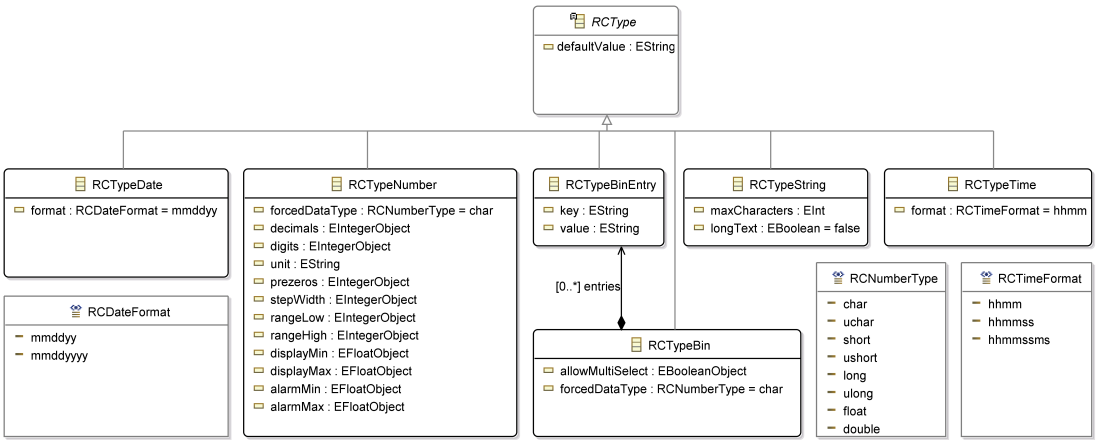


Figure 16.14: `Types` package of the simplified RC metamodel

Table 16.10: Metamodel element descriptions for figure 16.14

Concept	Description
`RCType`	Abstract Concept for the specification of datatypes in the radCase context
`RCTypeDate`	Concept for a special kind of datatype for date information
`RCTypeNumber`	Concept for a special kind of datatype for number information
`RCTypeBinEntry`	Concept for a special kind of datatype for binary information
`RCTypeString`	Concept for a special kind of datatype for string information
`RCTypeTime`	Concept for a special kind of datatype for time information
`RCDateFormat`	Enumeration concept specifying different types of date formats
`RCTypeBin`	Concept for a special kind of datatype for binary information including sets of `RCTypeBinEntry`
`RCNumberType`	Enumeration concept specifying multiple predefined number types
`RCTimeFormat`	Enumeration concept specifying different types of time formats

Finally, the `preprocessor` package is discussed in more detail.

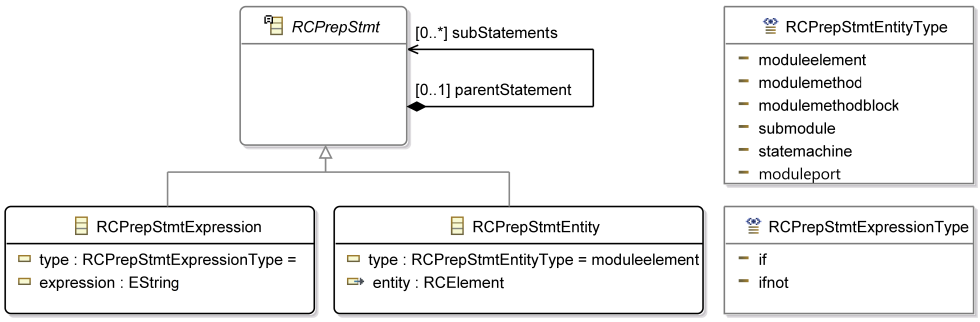


Figure 16.15: `Preprocessor` package of the simplified RC metamodel

Table 16.11: Metamodel element descriptions for figure 16.15

Concept	Description
`RCPrepStmt`	Abstract concept for the specification of preprocessor statements to determine the building blocks of the resulting target artifact
`RCPrepStmtEntityType`	Enumeration concept to determine the type of a building block encapsulated by a `RCPrepStmt`
`RCPrepStmtExpression`	Concept for the specification of a statement expression
`RCPrepStmtEntity`	Concept for the specification of a element affected by a preprocessor statement
`RCPrepStmtExpressionType`	Enumeration concept for the determination of a certain expression type

MBTSuite Metamodel Information

This section discusses the metamodel of test modeling in the context of the commercial tool *mbtSuite*. The metamodel explained in the following is a modified version of the conventional UML activity chart and extends it by test-specific aspects. How the model elements are connected is shown in figure 16.16 and then explained in detail in table 16.12.

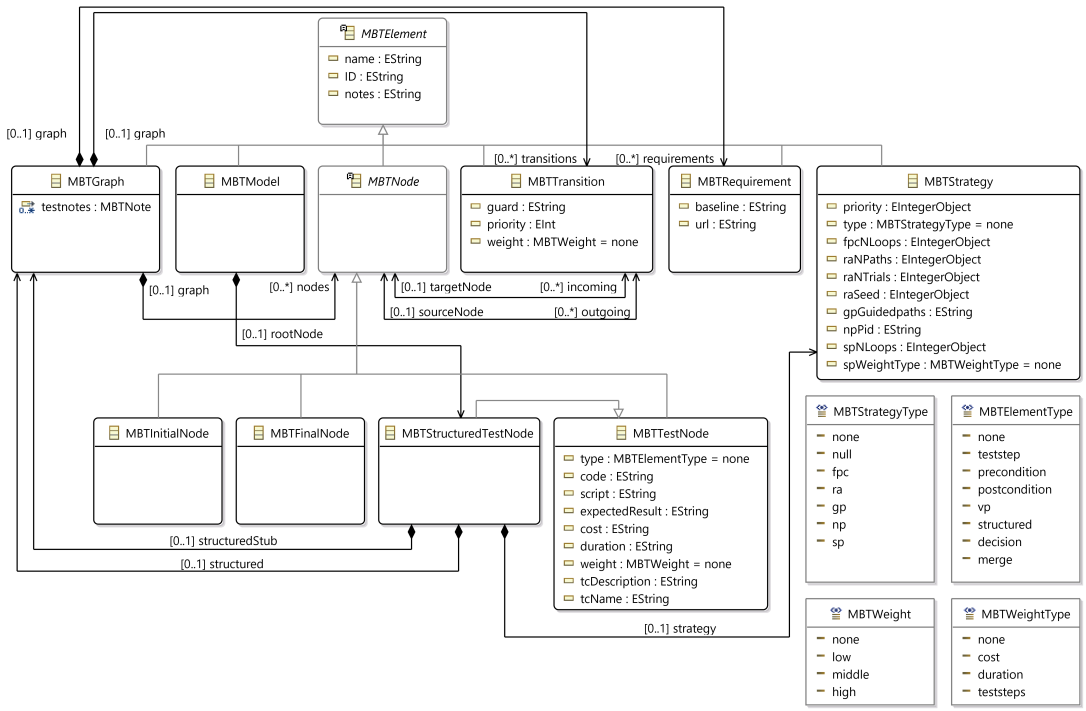


Figure 16.16: Simplified MBT metamodel

Table 16.12: Metamodel element descriptions for figure 16.16

Concept	Description
`MBTElement`	The most abstract concept of the mbt metamodel defining a unique identifier per element
`MBTGraph`	Concept capturing the elements on the same abstraction level and logical test unit
`MBTModel`	Concept representing the container for all graphs specified in a hierarchical test model
`MBTNode`	Abstract concept representing a node of the activity chart-like graph-based test model
`MBTTransiton`	Concept connecting two nodes of a test model graph
`MBTRequirement`	Concept encapsulating information about a test-specific requirement
`MBTStrategy`	Concept capturing the information for later applied test strategy which derives the set of concrete test cases from the hierarchical test model
`MBTInitialNode`	Concept for a special kind of node in the graph structure
`MBTFinalNode`	Concept for a special kind of node in the graph structure

Table 16.12: Metamodel element descriptions for figure 16.16 (continued)

`MBTStructuredNode`	Concept for a special node including a sub test model encapsulated in a `MBTGraph`
`MBTTestNode`	Concept for special kind of node specifying test instructions
`MBTStrategyType`	Enumeration concept for determining the type of applied strategy
`MBTElementType`	Enumeration concept for determining the type of test node
`MBTWeight`	Enumeration concept for specifying the `MBTTestNodes` weight
`MBTWeightType`	Enumeration concept for detailed specification of the weight type

FTA Metamodel Information

In this section all the metamodel concepts for a Fault Tree Analysis (FTA) are presented. At this point, a building structure of possible events is constructed, whose probabilities of occurrence are propagated to the root node, which in turn represents an abstract fault event. In the following, the connections between the concepts (figure 16.17) are dealt with in more detail, which are briefly explained in the context of table 16.13.

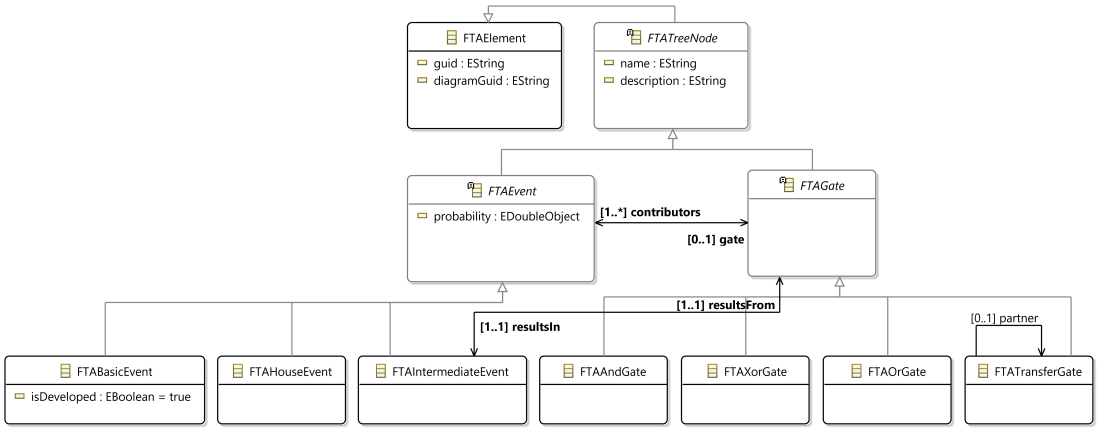


Figure 16.17: FTA metamodel

Table 16.13: Metamodel element descriptions for figure 16.17

Concept	**Description**
`FTAElement`	Most abstract concept giving each derived element a globally unique identifier
`FTATreeNode`	Abstract concept for elements taking part in the tree structure

Table 16.13: Metamodel element descriptions for figure 16.17 (continued)

`FTAEvent`	Abstract concept for elements representing events
`FTAGate`	Abstract concept for elements representing intermediate nodes which join child events according to a certain logic
`FTABasicEvent`	Concept representing the most basic version of a `FTAEvent`
`FTAHouseEvent`	Concept for elements representing intermediate events with binary probability
`FTAIntermediateEvent`	Concept for elements on intermediate levels of the tree structure
`FTAAndGate`	Concept representing a `AND` connection of subevents in the tree structure
`FTAXorGate`	Concept representing a `XOR` connection of subevents in the tree structure
`FTAOrGate`	Concept representing a `OR` connection of subevents in the tree structure
`FTATransferGate`	Concept for elements which connect to partial FTA trees

Additional Figures for the Elevator System

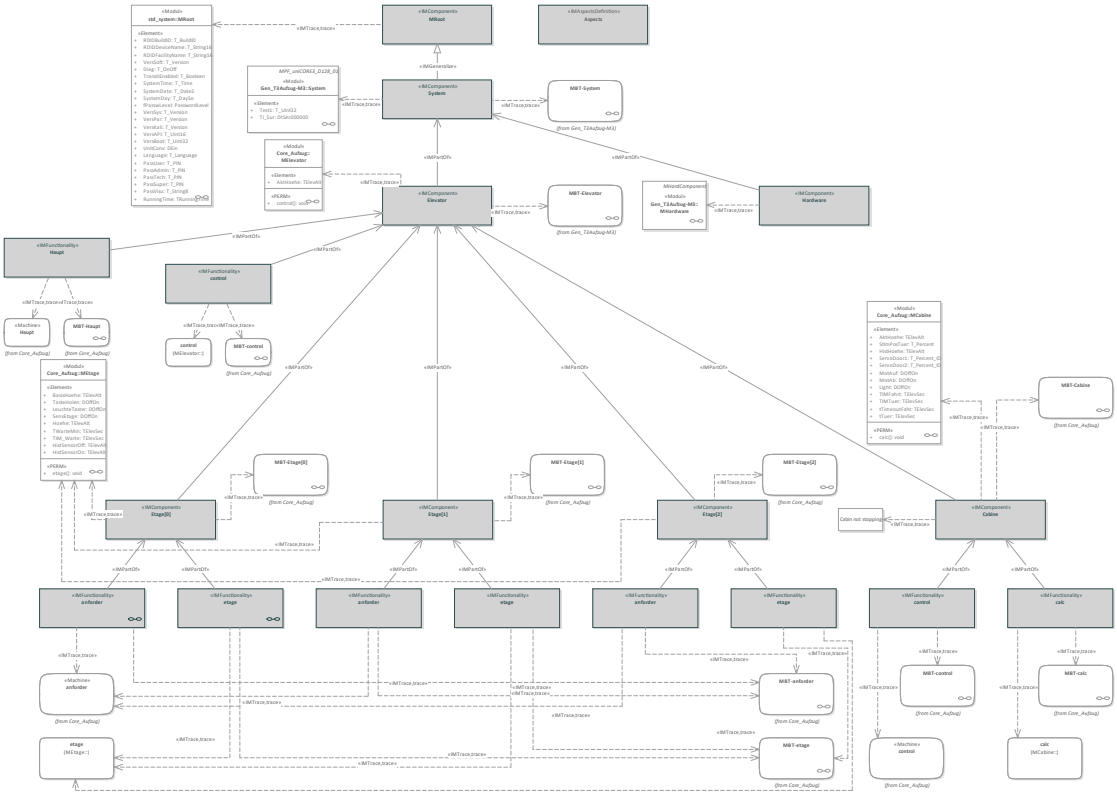


Figure 16.18: Overall Integration Model for the Elevator System

Additional Figures for the ALCS

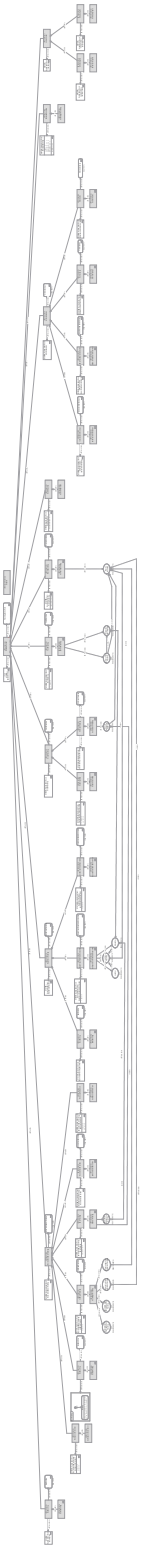


Figure 16.19: Overall Integration Model for the ALCS

Bibliography

[1] SysML 1.6. *OMG Systems Modeling Language (OMG SysML), Version 1.6*. Standard. Object Management Group, 2019.

[2] DO 178B/C. *Software Considerations in Airborne Systems and Equipment Certification*. Standard. RTCA, 1992.

[3] UML 2. *OMG Unified Modeling Language (OMG UML), Version 2.5.1*. Standard. Object Management Group, 2017.

[4] UTP 2. *UML Testing Profile 2 (UTP 2), Version 2.0*. Standard. Object Management Group (OMG), 2018.

[5] ISO 26262-6:2018. *Road vehicles — Functional safety — Part 6: Product development at the software level*. Standard. International Organization for Standardization, 2018.

[6] ISO 29119-1. *Software and systems engineering – Software testing – Part 1: Concepts and definitions*. Standard. International Organization for Standardization, 2013.

[7] ISO 29119-2. *Software and systems engineering – Software testing – Part 2: Test Processes*. Standard. International Organization for Standardization, 2013.

[8] ISO 29119-3. *Software and systems engineering – Software testing – Part 3: Test Documentation*. Standard. International Organization for Standardization, 2013.

[9] ISO 29119-4. *Software and systems engineering – Software testing – Part 4: Test Techniques*. Standard. International Organization for Standardization, 2015.

[10] ISO 29119-5. *Software and systems engineering – Software testing – Part 5: Keyword-Driven Testing*. Standard. International Organization for Standardization, 2016.

[11] IEC 62304:2006. *Medical device software — Software life cycle processes*. Standard. International Organization for Standardization, 2006.

[12] Muhammad Abbas et al. "Requirements dependencies-based test case prioritization for extra-functional properties". In: *2019 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. ICSTW '19. Xi'an, China: IEEE, 2019, pp. 159–163. DOI: 10.1109/ICSTW.2019.00045.

[13] Fredrik Abbors, Dragos Truscan, and Johan Lilius. "Tracing requirements in a model-based testing approach". In: *2009 First International Conference on Advances in System Testing and Validation Lifecycle*. VALID '09. Porto, Portugal: IEEE, 2009, pp. 123–128. DOI: 10.1109/VALID.2009.15.

[14] Anas Abuljadayel and Fadi Wedyan. "An approach for the generation of higher order mutants using genetic algorithms". In: *International Journal of Intelligent Systems and Applications* 10 (2018), p. 34.

[15] Ademar Aguiar and Gabriel David. "WikiWiki weaving heterogeneous software artifacts". In: *Proceedings of the 2005 international symposium on Wikis*. WikiSym '05. San Diego, California: Association for Computing Machinery, 2005, pp. 67–74. DOI: 10.1145/1104973.1104980.

[16] Iftekhar Ahmed et al. “Applying mutation analysis on kernel test suites: an experience report”. In: *2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. ICSTW 2017. Tokyo, Japan: IEEE, 2017, pp. 110–115. DOI: 10.1109/ICSTW.2017.26.

[17] ALF. *Action Language for Foundational UML (ALF), Version 1.1*. Standard. Object Management Group (OMG), 2017.

[18] Ståle Amland. “Risk-based testing:: Risk analysis fundamentals and metrics for software testing including a financial application case study”. In: *Journal of Systems and Software* 53 (2000), pp. 287–295. DOI: 10.1016/S0164-1212(00)00019-4.

[19] Paul Ammann and Jeff Offutt. *Introduction to software testing*. Cambridge University Press, 2016.

[20] Saswat Anand et al. “An orchestrated survey of methodologies for automated software test case generation”. In: *Journal of Systems and Software* 86 (2013), pp. 1978–2001. DOI: 10.1016/j.jss.2013.02.061.

[21] ANSI/IEEE 1008. *IEEE Standard for Software Unit Testing*. Standard. IEEE Computer Society Press, 1987.

[22] Larry Apfelbaum and John Doyle. “Model based testing”. In: *Software Quality Week Conference*. 1997, pp. 296–300.

[23] Md Junaid Arafeen and Hyunsook Do. “Test case prioritization using requirements-based clustering”. In: *2013 IEEE sixth international conference on software testing, verification and validation*. ICST ’13. Luxembourg, Luxembourg: IEEE, 2013, pp. 312–321. DOI: 10.1109/ICST.2013.12.

[24] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, et al. *Fundamental concepts of dependability*. University of Newcastle upon Tyne, Computing Science, 2001.

[25] James Bach. *Exploratory testing explained*. 2003.

[26] John Backus. “The history of Fortran I, II, and III”. In: *ACM Sigplan Notices* 13 (1978), pp. 165–180. DOI: 10.1145/960118.808380.

[27] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT press, 2008.

[28] Paul Baker, Shiou Loh, and Frank Weil. “Model-Driven engineering in a large industrial context—motorola case study”. In: *International Conference on Model Driven Engineering Languages and Systems*. MODELS 2005. Montego Bay, Jamaica: Springer Berlin Heidelberg, 2005, pp. 476–491. DOI: 10.1007/11557432_36.

[29] N Md Jubair Basha, Salman Abdul Moiz, and Mohammed Rizwanullah. “Model based software development: Issues & challenges”. In: *Special Issue of International Journal of Computer Science & Informatics (IJCSI)* 3 (2012), pp. 226–230. DOI: 10.47893/IJCSI.2013.1123.

[30] Ted L Bennett and Paul W Wennberg. “Eliminating embedded software defects prior to integration test”. In: *Crosstalk, Journal of Defence Software Engineering* (2005), pp. 13–18. DOI: 10.1.1.434.9838.

[31] Simona Bernardi, Susanna Donatelli, and José Merseguer. "From UML Sequence Diagrams and Statecharts to Analysable Petri Net Models". In: *Proceedings of the 3rd International Workshop on Software and Performance*. WOSP '02. Rome, Italy: Association for Computing Machinery, 2002, pp. 35–45. DOI: 10.1145/584369.584376.

[32] Donald J Berndt and Alison Watkins. "Investigating the performance of genetic algorithm-based software test case generation". In: *Eighth IEEE International Symposium on High Assurance Systems Engineering, 2004. Proceedings.* Tampa, FL, USA: IEEE, 2004, pp. 261–262. DOI: 10.1109/HASE.2004.1281750.

[33] Sami Beydeda, Matthias Book, Volker Gruhn, et al. *Model-driven software development*. Vol. 15. Springer, 2005.

[34] Jean Bézivin, Frédéric Jouault, and Patrick Valduriez. "On the need for megamodels". In: *Proceedings of the OOPSLA/GPCE: Best Practices for Model-Driven Software Development workshop, 19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*. Citeseer, 2004, pp. 1–9.

[35] Xavier Blanc, Marie-Pierre Gervais, and Prawee Sriplakich. "Model bus: Towards the interoperability of modelling tools". In: *Model driven architecture*. MDAFA 2003 and MDAFA 2004. Twente, The Netherlands: Springer Berlin Heidelberg, 2004, pp. 17–32. DOI: 10.1007/11538097_2.

[36] B. W. Boehm. "Software Engineering Economics". In: *IEEE Transactions on Software Engineering* SE-10 (1984), pp. 4–21. DOI: 10.1109/TSE.1984.5010193.

[37] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. "Model-driven software engineering in practice". In: *Synthesis Lectures on Software Engineering* 1 (2012), pp. 1–182. DOI: 10.2200/S00751ED2V01Y201701SWE004.

[38] Cécile Braunstein et al. *A SysML Test Model and Test Suite for the ETCS Ceiling Speed Monitor*. Technical Report. ITEA2 Project, 2014.

[39] 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 (2010), pp. 177–207. DOI: 10.1002/stvr.410.

[40] BS 7925-1. *Software testing. Vocabulary*. Standard. BSI British Standards, 1998.

[41] BS 7925-2. *Software testing. Software Component Testing*. Standard. BSI British Standards, 1998.

[42] Timothy A Budd and Dana Angluin. "Two notions of correctness and their relation to testing". In: *Acta informatica* 18 (1982), pp. 31–45. DOI: 10.1007/BF00625279.

[43] Federico Ciccozzi, Ivano Malavolta, and Bran Selic. "Execution of UML models: a systematic review of research and practice". In: *Software & Systems Modeling* 18 (2019), pp. 2313–2360. DOI: 10.1007/s10270-018-0675-4.

[44] Tony Clark, Paul Sammut, and James Willans. *Applied metamodelling: a foundation for language driven development*. Ceteva, 2008.

[45] Henry Coles et al. “Pit: a practical mutation testing tool for java”. In: *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ISSTA 2016. Saarbrücken, Germany: Association for Computing Machinery, 2016, pp. 449–452. DOI: 10.1145/2931037.2948707.

[46] Michelle L. Crane and Juergen Dingel. “Towards a UML Virtual Machine: Implementing an Interpreter for UML 2 Actions and Activities”. In: *Proceedings of the 2008 Conference of the Center for Advanced Studies on Collaborative Research: Meeting of Minds*. CASCON ’08. Ontario, Canada: Association for Computing Machinery, 2008, pp. 96–110. DOI: 10.1145/1463788.1463799.

[47] Krzysztof Czarnecki and Simon Helsen. “Feature-based survey of model transformation approaches”. In: *IBM Systems Journal* 45 (2006), pp. 621–645. DOI: 10.1147/sj.453.0621.

[48] James B Dabney and Thomas L Harman. *Mastering simulink*. Pearson, 2004.

[49] JB Dabney. “Return on Investment of Independent Verification and Validation Study Preliminary Phase 2B Report”. In: *Fairmont, WV: NASA IV&V Facility* (2003).

[50] Marcos Didonet Del Fabro and Patrick Valduriez. “Semi-Automatic Model Integration Using Matching Transformations and Weaving Models”. In: *Proceedings of the 2007 ACM Symposium on Applied Computing*. SAC ’07. Seoul, Korea: Association for Computing Machinery, 2007, pp. 963–970. DOI: 10.1145/1244002.1244215.

[51] Julien Delange et al. *Evaluating and Mitigating the Impact of Complexity in Software Models*. Tech. rep. CMU/SEI-2015-TR-013. Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2015. URL: http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=448083.

[52] Pedro Delgado-Pérez et al. “Assessment of class mutation operators for C++ with the MuCPP mutation system”. In: *Information and Software Technology* 81 (2017), pp. 169–184. DOI: 10.1016/j.infsof.2016.07.002.

[53] Richard A DeMillo, Richard J Lipton, and Frederick G Sayward. “Hints on test data selection: Help for the practicing programmer”. In: *Computer* 11 (1978), pp. 34–41. DOI: 10.1109/C-M.1978.218136.

[54] Anna Derezińska. “A quality estimation of mutation clustering in c# programs”. In: *Proceedings of the 8th International Conference on Dependability and Complex Systems, September 9-13, 2013*. DepCoS-RELCOMEX. Brunow, Poland: Springer International Publishing, 2013, pp. 119–129. DOI: 10.1007/978-3-319-00945-2_11.

[55] Gergely Dévai et al. “UML Model Execution via Code Generation.” In: *Proceedings of the 1st International Workshop on Executable Modeling co-located with ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS 2015*. MODELS ’15. Ottawa, Canada: CEUR-WS.org, 2015, pp. 9–15.

[56] Davide Di Ruscio, Romina Eramo, and Alfonso Pierantonio. "Model Transformations". In: *Formal Methods for Model-Driven Engineering: 12th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2012, Bertinoro, Italy, June 18-23, 2012. Advanced Lectures*. Ed. by Marco Bernardo, Vittorio Cortellessa, and Alfonso Pierantonio. Springer Berlin Heidelberg, 2012, pp. 91–136. DOI: 10.1007/978-3-642-30982-3_4.

[57] Arilo C Dias Neto et al. "A survey on model-based testing approaches: a systematic review". In: *Proceedings of the 1st ACM International Workshop on Empirical Assessment of Software Engineering Languages and Technologies: Held in Conjunction with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE) 2007*. WEASELTech '07. Atlanta, Georgia: Association for Computing Machinery, 2007, pp. 31–36. DOI: 10.1145/1353673.1353681.

[58] Edsger Wybe Dijkstra et al. *Notes on structured programming*. 1970.

[59] Dominik Anderle. "Funktionale Absicherung auf dem Weg zum autonomen Fahren am Beispiel eines Autobahnpiloten". MA Thesis. University Augsburg, 2017.

[60] Emelie Engström, Per Runeson, and Mats Skoglund. "A systematic review on regression test selection techniques". In: *Information and Software Technology* 52 (2010), pp. 14–30. DOI: 10.1016/j.infsof.2009.07.001.

[61] Eduard Paul Enoiu, Daniel Sundmark, and Paul Pettersson. "Model-based test suite generation for function block diagrams using the uppaal model checker". In: *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops*. ICSTW 2013. Luxembourg, Luxembourg: IEEE, 2013, pp. 158–167. DOI: 10.1109/ICSTW.2013.27.

[62] Sandra Camargo Pinto Ferraz Fabbri et al. "Mutation testing applied to validate specifications based on petri nets". In: *International Conference on Formal Techniques for Distributed Objects, Components, and Systems*. FORTE 1995. Montreal, Canada: Springer US, 1995, pp. 329–337. DOI: 10.1007/978-0-387-34945-9_24.

[63] Sandra Camargo Pinto Ferraz Fabbri et al. "Mutation testing applied to validate specifications based on statecharts". In: *Proceedings 10th International Symposium on Software Reliability Engineering (Cat. No. PR00443)*. ISSRE 1999. Boca Raton, FL, USA: IEEE, 1999, pp. 210–219. DOI: 10.1109/ISSRE.1999.809326.

[64] Peter H Feiler et al. *Reliability validation and improvement framework*. Tech. rep. Carnegie-Mellon Univ Pittsburgh Pa Software Engineering Inst, 2012.

[65] Michael Felderer and Ina Schieferdecker. "A taxonomy of risk-based testing". In: *International Journal on Software Tools for Technology Transfer* 16 (2014), pp. 559–568. DOI: 10.1007/s10009-014-0332-3.

[66] Michael Felderer, Marc-Florian Wendland, and Ina Schieferdecker. "Risk-based testing". In: *International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*. Springer Berlin Heidelberg, 2014, pp. 274–276. DOI: 10.1007/978-3-662-45231-8_19.

[67] MDA FM. *The MDA Foundation Model*. Standard. Object Management Group (OMG), 2010.

[68] Martin Fowler. *Domain-specific languages*. Pearson Education, 2010.

[69] Robert France and Bernhard Rumpe. "Model-driven Development of Complex Software: A Research Roadmap". In: *Future of Software Engineering*. FOSE '07. Minneapolis, MN, USA: IEEE Computer Society, 2007, pp. 37–54. DOI: 10.1109/FOSE.2007.14.

[70] fUML. *Semantics of a Foundational Subset for Executable UML Models (fUML), Version 1.4*. Standard. Object Management Group (OMG), 2018.

[71] Daniel Galin. *Software quality assurance: from theory to implementation*. Pearson Education India, 2004.

[72] Erich Gamma. *Design patterns: elements of reusable object-oriented software*. Pearson Education India, 1995.

[73] Gregory Gay. "Generating effective test suites by combining coverage criteria". In: *International Symposium on Search Based Software Engineering*. SSBSE 2017. Paderborn, Germany: Springer International Publishing, 2017, pp. 65–82. DOI: 10.1007/978-3-319-66299-2_5.

[74] Gregory Gay et al. "The risks of coverage-directed test case generation". In: *IEEE Transactions on Software Engineering* 41 (2015), pp. 803–819. DOI: 10.1109/TSE.2015.2421011.

[75] IMACS GmbH. *radCase - Model-Driven Generation*. 2020. URL: https://www.radcase.com/ (visited on 04/14/2021).

[76] Pablo Gómez-Abajo, Esther Guerra, and Juan de Lara. "Wodel: a domain-specific language for model mutation". In: *Proceedings of the 31st Annual ACM Symposium on Applied Computing*. SAC '16. Pisa, Italy: Association for Computing Machinery, 2016, pp. 1968–1973. DOI: 10.1145/2851613.2851751.

[77] Pablo Gómez-Abajo et al. "Wodel-Test: a model-based framework for language-independent mutation testing". In: *Software and Systems Modeling* (2020), pp. 1–27. DOI: 10.1007/s10270-020-00827-0.

[78] Rahul Gopinath, Carlos Jensen, and Alex Groce. "Topsy-Turvy: a smarter and faster parallelization of mutation analysis". In: *Proceedings of the 38th International Conference on Software Engineering Companion*. ICSE '16. Austin, Texas: Association for Computing Machinery, 2016, pp. 740–743. DOI: 10.1145/2889160.2892655.

[79] O. C. Z. Gotel and C. W. Finkelstein. "An analysis of the requirements traceability problem". In: *Proceedings of IEEE International Conference on Requirements Engineering*. Colorado Springs, CO, USA: IEEE, 1994, pp. 94–101. DOI: 10.1109/ICRE.1994.292398.

[80] Dorothy Graham, Erik Van Veenendaal, and Isabel Evans. *Foundations of software testing: ISTQB certification*. Cengage Learning EMEA, 2008.

[81] Jan Friso Groote, Tim WDM Kouters, and Ammar Osaiweran. "Specification guidelines to avoid the state space explosion problem". In: *Software Testing, Verification and Reliability* 25 (2015), pp. 4–33. DOI: 10.1002/stvr.1536.

[82] Juergen Grossmann, Diana Alina Serbanescu, and Ina Schieferdecker. "Testing Embedded Real Time Systems with TTCN-3." In: *2009 International Conference on Software Testing Verification and Validation*. ICST 2009. Denver, CO, USA: IEEE Computer Society, 2009, pp. 81–90. DOI: 10.1109/ICST.2009.37.

[83] Noël Hagemann, Reinhard Pröll, and Bernhard Bauer. "Towards abstract test execution in early stages of model-driven software development". In: *Proceedings of the 8th International Conference on Model-Driven Engineering and Software Development - Volume 1: MODELSWARD, February 25-27, 2020*. MODELSWARD 2020. Valletta, Malta: SciTePress, 2020, pp. 216–226. DOI: 10.5220/0008934802160226.

[84] David Harel and Michal Politi. *Modeling reactive systems with statecharts: the STATEMATE approach*. McGraw-Hill, Inc., 1998.

[85] M Jean Harrold, Rajiv Gupta, and Mary Lou Soffa. "A methodology for controlling the size of a test suite". In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 2 (1993), pp. 270–285. DOI: 10.1145/152388.152391.

[86] Regina Hebig, Andreas Seibel, and Holger Giese. "On the unification of megamodels". In: *Electronic Communications of the EASST* 42 (2012). DOI: 10.14279/tuj.eceasst.42.704.

[87] Mats PE Heimdahl and Devaraj George. "Test-suite reduction for model based tests: Effects on test quality and implications for testing". In: *Proceedings. 19th International Conference on Automated Software Engineering, 2004.* ASE '04. Linz, Austria: IEEE, 2004, pp. 176–185. DOI: 10.1109/ASE.2004.1342735.

[88] Hadi Hemmati et al. "An enhanced test case selection approach for model-based testing: an industrial case study". In: *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*. FSE '10. Santa Fe, New Mexico, USA: Association for Computing Machinery, 2010, pp. 267–276. DOI: 10.1145/1882291.1882331.

[89] Kim Herzig et al. "The art of testing less without sacrificing quality". In: *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. ICSE '15. Florence, Italy: IEEE, 2015, pp. 483–493. DOI: 10.1109/ICSE.2015.66.

[90] Anders Hessel et al. "Time-optimal real-time test case generation using UPPAAL". In: *International Workshop on Formal Approaches to Software Testing*. FATES 2003. Montreal, Quebec, Canada: Springer Berlin Heidelberg, 2003, pp. 114–130. DOI: 10.1007/978-3-540-24617-6_9.

[91] Robert M Hierons and Mercedes G Merayo. "Mutation testing from probabilistic and stochastic finite state machines". In: *Journal of Systems and Software* 82 (2009), pp. 1804–1818. DOI: 10.1016/j.jss.2009.06.030.

[92] Rich Hilliard. "Using the UML for architectural description". In: *International Conference on the Unified Modeling Language*. UML '99. Fort Collins, CO, USA: Springer Berlin Heidelberg, 1999, pp. 32–48. DOI: 10.1007/3-540-46852-8_4.

[93] Dirk W Hoffmann. "Software-Verifikation". In: *Software-Qualität*. Springer Berlin Heidelberg, 2008, pp. 333–369. DOI: 10.1007/978-3-540-76323-9_6.

[94] T. Honglei, S. Wei, and Z. Yanan. "The Research on Software Metrics and Software Complexity Metrics". In: *2009 International Forum on Computer Science-Technology and Applications*. IFCSTA '09. Chongqing, China: IEEE, 2009, pp. 131–136. DOI: 10.1109/IFCSTA.2009.39.

[95] William E. Howden. "Weak mutation testing and completeness of test sets". In: *IEEE Transactions on Software Engineering* 4 (1982), pp. 371–379. DOI: 10.1109/TSE.1982.235571.

[96] Antti Huima. "Implementing conformiq qtronic". In: *Testing of Software and Communicating Systems*. FATES 2007. Tallinn, Estonia: Springer Berlin Heidelberg, 2007, pp. 1–12. DOI: 10.1007/978-3-540-73066-8_1.

[97] Hussain, Shamaila. "Mutation clustering". MA Thesis. Kings College London, Strand, London, 2008.

[98] Laura Inozemtseva and Reid Holmes. "Coverage is not strongly correlated with test suite effectiveness". In: *Proceedings of the 36th international conference on software engineering*. ICSE 2014. Hyderabad, India: Association for Computing Machinery, 2014, pp. 435–445. DOI: 10.1145/2568225.2568271.

[99] Jean-Marc Jézéquel. "Model driven design and aspect weaving". In: *Software & Systems Modeling* 7 (2008), pp. 209–218. DOI: 10.1007/s10270-008-0080-5.

[100] Y. Jia and M. Harman. "An Analysis and Survey of the Development of Mutation Testing". In: *IEEE Transactions on Software Engineering* 37 (2011), pp. 649–678. DOI: 10.1109/TSE.2010.62.

[101] Johannes Kübel. "Model-to-Model Transformationen im Kontext modell-basierter Software- und System-Analysen". BA Thesis. University Augsburg, 2020.

[102] Johannes Kühbacher. "Model-to-Model Transformationen zur Adaption Modellzentrischer Testmechanismen am Beispiel U2TP". BA Thesis. University Augsburg, 2020.

[103] Paul C Jorgensen. *Software testing: a craftsman's approach*. Auerbach Publications, 2014.

[104] A. Jossic et al. "Model Integration with Model Weaving: a Case Study in System Architecture". In: *2007 International Conference on Systems Engineering and Modeling*. MBSE '07. Haifa, Israel: IEEE, 2007, pp. 79–84. DOI: 10.1109/ICSEM.2007.373336.

[105] Frédéric Jouault et al. "Inter-DSL Coordination Support by Combining Megamodeling and Model Weaving". In: *Proceedings of the 2010 ACM Symposium on Applied Computing*. SAC '10. Sierre, Switzerland: Association for Computing Machinery, 2010, pp. 2011–2018. DOI: 10.1145/1774088.1774511.

[106] René Just and Franz Schweiggert. "Higher accuracy and lower run time: efficient mutation analysis using non-redundant mutation operators". In: *Software Testing, Verification and Reliability* 25 (2015), pp. 490–507. DOI: 10.1002/stvr.1561.

[107] J Kamga, J Herrmann, and P Joshi. *D-MINT automotive case study*. Deliverable. ITEA2 Project, 2007.

[108] Muhammad Khatibsyarbini et al. "Test case prioritization approaches in regression testing: A systematic literature review". In: *Information and Software Technology* 93 (2018), pp. 74–93. DOI: 10.1016/j.infsof.2017.08.014.

[109] Sarfraz Khurshid, Corina S Păsăreanu, and Willem Visser. "Generalized symbolic execution for model checking and testing". In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. TACAS 2003. Warsaw, Poland: Springer Berlin Heidelberg, 2003, pp. 553–568. DOI: 10.1007/3-540-36577-X_40.

[110] James C King. “Symbolic execution and program testing”. In: *Communications of the ACM* 19 (1976), pp. 385–394. DOI: 10.1145/360248.360252.

[111] Marinos Kintis et al. “Detecting trivial mutant equivalences via compiler optimisations”. In: *IEEE Transactions on Software Engineering* 44 (2017), pp. 308–333. DOI: 10.1109/TSE.2017.2684805.

[112] Andrei Kirshin, Dolev Dotan, and Alan Hartman. “A UML simulator based on a generic model execution engine”. In: *Proceedings of the 2006 International Conference on Models in Software Engineering*. MoDELS’06. Genoa, Italy: Springer-Verlag, 2006, pp. 324–326. DOI: 10.5555/1762828.1762882.

[113] Anneke G Kleppe et al. *MDA explained: the model driven architecture: practice and promise*. Addison-Wesley Professional, 2003.

[114] Hermann Kopetz. “The complexity challenge in embedded system design”. In: *2008 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*. ISORC ’08. Orlando, FL, USA: IEEE, 2008, pp. 3–12. DOI: 10.1109/ISORC.2008.14.

[115] Willibald Krenn et al. “Momut:: UML model-based mutation testing for UML”. In: *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*. ICST ’15. Graz, Austria: IEEE, 2015, pp. 1–8. DOI: 10.1109/ICST.2015.7102627.

[116] Markus Kusano and Chao Wang. “CCmutator: A mutation generator for concurrency constructs in multithreaded C/C++ applications”. In: *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. ASE 2013. Silicon Valley, CA, USA: IEEE, 2013, pp. 722–725. DOI: 10.1109/ASE.2013.6693142.

[117] Janusz W. Laski and Bogdan Korel. “A data flow oriented program testing strategy”. In: *IEEE Transactions on Software Engineering* SE-9 (1983), pp. 347–354. DOI: 10.1109/TSE.1983.236871.

[118] Linus Daniel Straub. “Abstract Execution of Graph-based Test Descriptions in Model-Driven Software Development”. MA Thesis. University Augsburg, 2018.

[119] Zhifang Liu, Xiaopeng Gao, and Xiang Long. “Adaptive random testing of mobile application”. In: *2010 2nd International Conference on Computer Engineering and Technology*. ICCET ’10. Chengdu, China: IEEE, 2010, pp. V2-297-V2–301. DOI: 10.1109/ICCET.2010.5485442.

[120] Yu-Seung Ma and Sang-Woon Kim. “Mutation testing cost reduction by clustering overlapped mutants”. In: *Journal of Systems and Software* 115 (2016), pp. 18–30. DOI: 10.1016/j.jss.2016.01.007.

[121] Yu-Seung Ma, Yong-Rae Kwon, and Jeff Offutt. “Inter-class mutation operators for Java”. In: *13th International Symposium on Software Reliability Engineering, 2002. Proceedings.* ISSRE 2002. Annapolis, MD, USA: IEEE, 2002, pp. 352–363. DOI: 10.1109/ISSRE.2002.1173287.

[122] Yu-Seung Ma, Jeff Offutt, and Yong-Rae Kwon. “MuJava: a mutation system for Java”. In: *Proceedings of the 28th international conference on Software engineering*. ICSE ’06. Shanghai, China: Association for Computing Machinery, 2006, pp. 827–830. DOI: 10.1145/1134285.1134425.

[123] Aditya P Mathur and W Eric Wong. "An empirical comparison of data flow and mutation-based test adequacy criteria". In: *Software Testing, Verification and Reliability* 4 (1994), pp. 9–31. DOI: 10.1002/stvr.4370040104.

[124] Tanja Mayerhofer. "Testing and debugging UML models based on fUML". In: *2012 34th International Conference on Software Engineering (ICSE)*. ICSE '12. Zurich, Switzerland: IEEE, 2012, pp. 1579–1582. DOI: 10.1109/ICSE.2012.6227032.

[125] Tanja Mayerhofer and Philip Langer. "Moliz: A model execution framework for UML models". In: *Proceedings of the 2nd International Master Class on Model-Driven Engineering: Modeling Wizards*. MW '12. Innsbruck, Austria: Association for Computing Machinery, 2012, pp. 1–2. DOI: 10.1145/2448076.2448079.

[126] MDA. *Model Driven Architecture (MDA), MDA Guide rev. 2.0*. Standard. Object Management Group (OMG), 2014.

[127] Tom Mens and Pieter Van Gorp. "A taxonomy of model transformation". In: *Electronic Notes in Theoretical Computer Science* 152 (2006), pp. 125–142. DOI: 10.1016/j.entcs.2005.10.021.

[128] Nasir Mehmood Minhas et al. "A systematic mapping of test case generation techniques using UML interaction diagrams". In: *Journal of Software: Evolution and Process* 32 (2020). DOI: 10.1002/smr.2235.

[129] MOF. *About The Meta Object Facility Specification, Version 2.5.1*. Standard. Object Management Group (OMG), 2016.

[130] Mark Mossberg et al. "Manticore: A user-friendly symbolic execution framework for binaries and smart contracts". In: *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. ASE 2019. San Diego, CA, USA: IEEE, 2019, pp. 1186–1189. DOI: 10.1109/ASE.2019.00133.

[131] Noël Hagemann. "Datenfluss-basierte abstrakte Testausführung in der Modell-basierten Softwareentwicklung". MA Thesis. University Augsburg, 2019.

[132] OASIS Open Project. *Open Services for Lifecycle Collaboration*. 2020. URL: https://open-services.net/ (visited on 04/14/2021).

[133] A Jefferson Offutt, Gregg Rothermel, and Christian Zapf. "An experimental evaluation of selective mutation". In: *Proceedings of 1993 15th International Conference on Software Engineering*. ICSE 1993. Baltimore, MD, USA: IEEE, 1993, pp. 100–107. DOI: 10.1109/ICSE.1993.346062.

[134] A Jefferson Offutt and Roland H Untch. "Mutation 2000: Uniting the orthogonal". In: *Mutation testing for the new century*. Springer US, 2001, pp. 34–44. DOI: 10.1007/978-1-4757-5939-6_7.

[135] Mike Papadakis et al. "Mutation testing advances: an analysis and survey". In: *Advances in Computers Vol 112*. Elsevier, 2019, pp. 275–378. DOI: 10.1016/bs.adcom.2018.03.015.

[136] Jan Peleska et al. "A real-world benchmark model for testing concurrent real-time systems in the automotive domain". In: *IFIP International Conference on Testing Software and Systems*. ICTSS 2011. Paris, France: Springer Berlin Heidelberg, 2011, pp. 146–161. DOI: 10.1007/978-3-642-24580-0_11.

[137] Jan Peleska et al. *Turn indicator model overview*. Technical Report. University of Bremen, 2014.

[138] Goran Petrovic et al. "An industrial application of mutation testing: Lessons, challenges, and research directions". In: *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. ICSTW 2018. Västerås, Sweden: IEEE, 2018, pp. 47–53. DOI: 10.1109/ICSTW.2018.00027.

[139] Hoang Pham. *Software reliability*. Springer Science & Business Media, 2000.

[140] Alessandro Viola Pizzoleto et al. "A systematic literature review of techniques and metrics to reduce the cost of mutation testing". In: *Journal of Systems and Software* 157 (2019). DOI: 10.1016/j.jss.2019.07.100.

[141] Strategic Planning. "The economic impacts of inadequate infrastructure for software testing". In: *National Institute of Standards and Technology* (2002).

[142] Macario Polo, Mario Piattini, and Ignacio Garcia-Rodriguez. "Decreasing the cost of mutation testing with second-order mutants". In: *Software Testing, Verification and Reliability* 19 (2009), pp. 111–131. DOI: 10.1002/stvr.392.

[143] Alexander Pretschner and Jan Philipps. "10 methodological issues in model-based testing". In: *Model-Based Testing of Reactive Systems: Advanced Lectures*. Springer Berlin Heidelberg, 2005, pp. 281–291. DOI: 10.1007/11498490_13.

[144] Alexander Pretschner et al. "One evaluation of model-based testing and its automation". In: *Proceedings of the 27th international conference on Software engineering*. ICSE '05. St. Louis, MO, USA: Association for Computing Machinery, 2005, pp. 392–401. DOI: 10.1145/1062455.1062529.

[145] Reinhard Pröll and Bernhard Bauer. "A model-based test case management approach for integrated sets of domain-specific models". In: *Proceedings of the 2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), 9-13 April 2018*. ICSTW 2018. Västerås, Sweden: IEEE, 2018, pp. 175–184. DOI: 10.1109/icstw.2018.00048.

[146] Reinhard Pröll and Bernhard Bauer. "Toward a consistent and strictly model-based interpretation of the ISO/IEC/IEEE 29119 for early testing activities". In: *Proceedings of the 6th International Conference on Model-Driven Engineering and Software Development, January 22-24, 2018*. AMARETTO 2018. Funchal, Madeira, Portugal: SciTePress, 2018, pp. 699–706. DOI: 10.5220/0006749606990706.

[147] Reinhard Pröll, Noël Hagemann, and Bernhard Bauer. "Abstract Test Execution for Early Testing Activities in Model-Driven Scenarios". In: *Communications in Computer and Information Science* 1349 (2021), pp. 273–297. DOI: 10.1007/978-3-030-67445-8_12.

[148] Reinhard Pröll, Adrian Rumpold, and Bernhard Bauer. "Applying integrated domain-specific modeling for multi-concerns development of complex systems". In: *Communications in Computer and Information Science* 880 (2018), pp. 247–271. DOI: 10.1007/978-3-319-94764-8_11.

[149] Sandra Rapps and Elaine J. Weyuker. "Selecting software test data using data flow information". In: *IEEE transactions on software engineering* SE-11 (1985), pp. 367–375. DOI: 10.1109/TSE.1985.232226.

[150] Sanjai Rayadurgam and Mats Per Erik Heimdahl. "Coverage based test-case generation using model checkers". In: *Proceedings of the Eighth Annual IEEE International Conference and Workshop On the Engineering of Computer-Based Systems-ECBS 2001*. Washington, DC, USA: IEEE, 2001, pp. 83–91. DOI: 10.1109/ECBS.2001.922409.

[151] Adrian Rumpold, Reinhard Pröll, and Bernhard Bauer. "A domain-aware framework for integrated model-based system analysis and design". In: *Proceedings of the 5th International Conference on Model-Driven Engineering and Software Development: February 19-21, 2017*. MODELSWARD 2017. Porto, Portugal: SciTePress, 2017, pp. 157–168. DOI: 10.5220/0006206301570168.

[152] Christian Saad. "Data-flow based Model Analysis: Approach, Implementation and Applications". doctoralthesis. Universität Augsburg, 2015.

[153] Alberto Sangiovanni-Vincentelli, Werner Damm, and Roberto Passerone. "Taming Dr. Frankenstein: Contract-based design for cyber-physical systems". In: *European journal of control* 18 (2012), pp. 217–238. DOI: 10.3166/ejc.18.217-238.

[154] Ina Schieferdecker et al. "The UML 2.0 testing profile and its relation to TTCN-3". In: *IFIP International Conference on Testing of Software and Communicating Systems*. TestCom 2003. Sophia Antipolis, France: Springer Berlin Heidelberg, 2003, pp. 79–94. DOI: 10.1007/3-540-44830-6_7.

[155] Bran Selic. "The pragmatics of model-driven development". In: *IEEE software* 20 (2003), pp. 19–25. DOI: 10.1109/MS.2003.1231146.

[156] sepp.med gmbh. *MBTsuite - The Testing Framework*. 2018. URL: https://mbtsuite.com (visited on 04/14/2021).

[157] Mahesh Shirole and Rajeev Kumar. "UML behavioral model based test case generation: a survey". In: *ACM SIGSOFT Software Engineering Notes* 38 (2013), pp. 1–13. DOI: 10.1145/2492248.2492274.

[158] D. I. De Silva et al. "Applicability of three cognitive complexity metrics". In: *Computer Science Education (ICCSE), 2013 8th International Conference on*. ICCSE '13. Colombo, Sri Lanka: IEEE, 2013, pp. 573–578. DOI: 10.1109/ICCSE.2013.6553975.

[159] Herbert Stachowiak. *Allgemeine Modelltheorie*. Springer, 1973.

[160] T. Stahl and M. Völter. *Modellgetriebene Softwareentwicklung: Techniken, Engineering, Management*. dpunkt-Verlag, 2005. ISBN: 9783898643108. URL: https://books.google.de/books?id=MKb6AAAACAAJ.

[161] Dominic Steinhöfel and Reiner Hähnle. "Abstract execution". In: *International Symposium on Formal Methods*. FM2019. Porto, Portugal: Springer International Publishing, 2019, pp. 319–336. DOI: 10.1007/978-3-030-30942-8_20.

[162] Santosh Kumar Swain, Durga Prasad Mohapatra, and Rajib Mall. "Test case generation based on use case and sequence diagram". In: *International Journal of Software Engineering* 3 (2010), pp. 21–52.

[163] Luay Ho Tahat et al. "Requirement-based automated black-box test generation". In: *25th Annual International Computer Software and Applications Conference. COMPSAC 2001*. COMPSAC 2001. Chicago, IL, USA: IEEE, 2001, pp. 489–495. DOI: 10.1109/CMPSAC.2001.960658.

[164] Jérémie Tatibouët et al. “Formalizing execution semantics of UML profiles with fUML models”. In: *International Conference on Model Driven Engineering Languages and Systems*. MODELS 2014. Valencia, Spain: Springer International Publishing, 2014, pp. 133–148. DOI: 10.1007/978-3-319-11653-2_9.

[165] University of Augsburg, IMACS GmbH. *Modellgetriebene Software Entwicklung für Funktionale Sicherheit von Automatisierungslösungen (MDSD4SiL)*. ZIM KF-Projekt KF 2751303LT4. Zentrales Innovationsprogramm Mittelstand (ZIM) des Bundesministeriums für Wirtschaft und Energie (BMWi) - Fördermodul Kooperationsprojekte, 2014.

[166] University of Augsburg, IMACS GmbH, AFRA GmbH. *Reduction of Test Complexity (ReTeC)*. ZIM KOOP NKF 16KN044120. Ganzheitliche modellbasierte Entwicklung und Test von eingebetteten Systemen. Zentrales Innovationsprogramm Mittelstand (ZIM) des Bundesministeriums für Wirtschaft und Energie (BMWi) - Kooperationsnetzwerke, 2015.

[167] University of Augsburg, IMACS GmbH, Razorcat Development GmbH, Viconnis Test Technologie GmbH, XITASO GmbH. *Test The Test (T3)*. ZIM KOOP NKF 16KN044137. Metriken zur Ermittlung der Testqualität, Testergebnismodelle und Lernalgorithmen. Zentrales Innovationsprogramm Mittelstand (ZIM) des Bundesministeriums für Wirtschaft und Energie (BMWi) - Kooperationsnetzwerke, 2018.

[168] Mark Utting and Bruno Legeard. *Practical model-based testing: a tools approach*. Elsevier, 2010.

[169] Mark Utting, Alexander Pretschner, and Bruno Legeard. “A taxonomy of model-based testing approaches”. In: *Software testing, verification and reliability* 22 (2012), pp. 297–312. DOI: 10.1002/stvr.456.

[170] EV Veenendaal. *Standard Glossary of Terms used in Software Testing - Version 3.3*. Glossary. International Software Testing Qualification Board, 2019.

[171] Eelco Visser. “A survey of rewriting strategies in program transformation systems”. In: *Electronic Notes in Theoretical Computer Science* 57 (2001), pp. 109–143. DOI: 10.1016/S1571-0661(04)00270-1.

[172] Elaine J Weyuker. “On testing non-testable programs”. In: *The Computer Journal* 25 (1982), pp. 465–470. DOI: 10.1093/comjnl/25.4.465.

[173] T. W. Williams and K. P. Parker. “Design for testability—A survey”. In: *Proceedings of the IEEE* 71 (1983), pp. 98–112. DOI: 10.1109/PROC.1983.12531.

[174] Mario Winter et al. *Basiswissen modellbasierter Test: Aus-und Weiterbildung zum ISTQB® Foundation Level–Certified Model-Based Tester*. dpunkt. verlag, 2016.

[175] W Eric Wong and Aditya P Mathur. “Reducing the cost of mutation testing: An empirical study”. In: *Journal of Systems and Software* 31 (1995), pp. 185–196. DOI: 10.1016/0164-1212(94)00098-0.

[176] Weichen Eric Wong. “On mutation and data flow”. PhD thesis. Purdue University West Lafayette, Indiana, 1993.

[177] MR Woodward and K Halewood. “From weak to strong, dead or alive? an analysis of some mutation testing issues”. In: *Workshop on software testing, verification, and analysis*. Banff, AB, Canada: IEEE, 1988, pp. 152–153. DOI: 10.1109/WST.1988.5370.

[178] Justyna Zander et al. “From U2TP models to executable tests with TTCN-3-an approach to model driven testing”. In: *IFIP International Conference on Testing of Communicating Systems*. TestCom 2005. Montreal, QC, Canada: Springer Berlin Heidelberg, 2005, pp. 289–303. DOI: 10.1007/11430230_20.

Glossary

A3F Architecture And Analysis Framework.

ALCS Automotive Light Control System.

ALF Action Language for Foundational UML.

ANSI American National Standards Institute.

AOM Aspect-Oriented Modeling.

AOP Aspect-Oriented Programming.

ASIL Automotive Safety Integrity Level.

ATE Abstract Test Execution.

ATL Atlas Transformation Language.

BFS Breadth First Search.

BSI British Standards Institute.

CBD Contract-Based Design.

CE Coupling Effect.

CFA-ATE Control Flow Aware Abstract Test Execution.

CIM Computation Independent Model.

CMOF Complete MOF.

CPH Competent Programmer Hypothesis.

CSM Ceiling Speed Monitor.

DevOps Development and Operations.

DFA DataFlow Analysis.

DFA-ATE Data Flow Aware Abstract Test Execution.

DFS Depth First Search.

DFT Design For Testability.

DSL Domain-Specific Language.

DSML Domain-Specific Modeling Language.

EGPP Execution Graph++.

EGPPMM Execution Graph++ Metamodel.

EMOF Essential MOF.

ETCS European Train Control System.

EVC European Vital Computer.

FMEA Failure Modes And Effects Analysis.

FTA Fault Tree Analysis.

fUML Foundational Subset for Executable UML Models.

GPML General Purpose Modeling Language.

IDL Interface Definition Language.

IEC International Electrotechnical Commission.

IEEE Institute of Electrical and Electronics Engineers.

IM Integration Model.

ISM Implementation Specific Model.

ISO International Organization for Standardization.

ISTQB International Software Testing Qualification Board.

JSON JavaScript Object Notation.

KPI Key Performance Indicator.

M2CT Model-to-Code Transformation.

M2MT Model-to-Model Transformation.

M2TT Model-to-Text Transformation.

MAF Model Analysis Framework.

MAS Mutation Adequacy Score.

MBSD Model-Based Software Development.

MBT Model-Based Testing.

MC/DC Modified Condition/Decision Coverage.

MCSD Model-Centric Software Development.

MCSTLC Model-Centric Software Testing Life Cycle.

MCT Model-Centric Testing.

MDA Model-Driven Architecture.

MDE Model-Driven Engineering.

MDSD Model-Driven Software Development.

MDT Model-Driven Testing.

MM Metamodel.

MMM Metametamodel.

MOF Meta Object Facility.

MOT Model-Oriented Testing.

MTDD Model-Level Test Driven Development.

OCL Object Constraint Language.

OMG Object Management Group.

OOAS Object Oriented Action System.

ORM Object Relational Mapper.

OSLC Open Services for Life Cycle Collaboration.

OTS Organizational Test Specification.

PIM Platform Independent Model.

PSD Purpose-Specific Data.

PSM Platform Specific Model.

QVT Query View Transformation.

QVTO Query View Transformation Operational.

RBT Risk-Based Testing.

SDLC Software Development Life Cycle.

SEI Software Engineering Institute.

STLC Software Testing Life Cycle.

SUD System Under Development.

SUT System Under Test.

SYSML Systems Modeling Language.

TCG Test Case Generation.

TCMAS Test Case Mutation Adequacy Score.

TDD Test-Driven Development.

TM Test Model.

TMS Test Model Scoping.

TTCN-3 Testing and Test Control Notation.

UML Unified Modeling Language.

UTP UML Testing Profile.

V&V Verification and Validation.

VIATRA VIsual Automated model TRAnsformation.

VMI Variable Modifying Instruction.

VVI Variable Verifying Instruction.