(ITEA 2 – 13017)

Enabling of Results from AMALTHEA and others
for Transfer into Application and
building Community around

Deliverable: D 3.1

Analysis of state of the art V&V techniques

Work Package: 3
Verification and Test

Task: 3.1
Analysis of state of the art V&V techniques

Document Type:	Deliverable	**Classification:**	Internal
Document Version:	Final	**Contract Start Date:**	01.09.2014
Document Preparation Date:	30.04.2015	**Duration:**	31.08.2017

History

Rev.	Content	Resp. Partner	Date
0.0	Draft Version	Izaskun de la Torre	11.03.2015
0.1	Transfer of Contents from Draft Version to Project Template	Jan Jatzkowski	07.04.2015
0.2	Contributions to chapters 2.1-2.6 and 3	Ainhoa Gracia / Urko Acosta	11.05.2015
0.3	First Version of Chapter 2.8 Product Line Analysis	Christopher Brink	15.05.2015
0.4	First Version of Section 3.4.6 MechatronicUML	David Schmelter	19.05.2015

Final Approval	Name	Partner

Contents

List of Figures

List of Tables

Summary

This document presents a detailed overview of the state of the art of model verification and validation techniques. On the one hand, this document describes the main model verification and validation techniques and methods: informal, static, dynamic, and formal methods, test case generation and validation techniques, simulation techniques and product line methodology. On the other hand, this document presents the main tools used in the model verification regarding to the methods and guidelines detailed above.

1 Introduction

In order to integrate existing industrial and academic verification tools to formally define and develop a new model-based verification approach to prove the correctness of software in a multi/many core environment, improving existing simulation techniques and test tools to support a thorough V&V of multi-core systems in a model-driven development environment, it's necessary a good understanding of the existing model verification and validation methods. On the one hand, the following V&V technique methods can be highlighted: informal, static, dynamic and formal methods. Advantages and disadvantages of each one are presented. Test case generation and validation techniques are analysed next. Finally, simulation techniques, including discrete-event simulation and instruction set simulation, and product line methodologies are explained.

On the other hand, a set of V&V tools is over viewed. These tools complement the previously analysed V&V techniques. In this way, it is aimed to integrate new or proven techniques and approaches for verification, simulation, and testing into the Amalthea toolchain.

2 V&V Guidelines, Techniques, Methods and Best Practice

2.1 Introduction to model verification and validation methods

Verification can be understood as the process of determining that a model implementation and its associated data accurately represents the developer's conceptual description and specification. Verification also evaluates the extent to which the model or simulation has been developed using sound and established software engineering techniques [83]. In general terms, software engineering methods will apply and the process will answer to the question 'Is the model coded right?'. Verification techniques are deployed to ensure that the requirements for the simulation model and its conceptual model design have been transformed into the computer model with sufficient accuracy. In other words, the verification phase of V&V focuses on comparing the elements of a simulation model of the system with the description of what the requirements and capabilities of the model were to be. Verification is an iterative process aimed at determining whether the product of each step in the development of the simulation model fulfils all the requirements levied on it by the previous step and is internally complete, consistent, and correct enough to support the next phase.[78]

On the other hand, the definition of validation involves the process of determining the degree to which a model or simulation and its associated data are an accurate representation of the real world from the perspective of the intended uses of the model or simulation [83]. Specific validation methods will be needed and the process will answer to the question 'Is the right model coded?'. Validation techniques are deployed to reach an acceptable level of confidence that the simulation model's results are sufficiently accurate for its intended use and applicable to the real-world system being modeled. The validation phase of V&V focuses on comparing the observed behaviour of elements of a system with the corresponding elements of a simulation model of the system, and on determining whether the differences are acceptable given the intended use of the model. If agreement is not obtained, the model is adjusted in order to bring it in closer agreement with the observed behaviour of the actual system (or errors in observation/experimentation or reference models/analyses are identified and rectified). [78]

The two terms are often used together and incorrectly treated as if they were interchangeable. Another common misconception is that V&V is synonymous with testing. V&V does not replace testing, nor does it include testing. V&V, when used properly, can determine if testing has been performed correctly. Testing is an important activity in all software life cycles. V&V, while not normally a life-cycle activity, makes sure that all lifecycle activities have been correctly performed (including testing). The V&V of modeling and simulation (M&S) is also different from V&V of other software artifacts. In M&S, the V&V is used to show that the software model is a useful representation of the real world.

The basic goals of model V&V are to demonstrate that the requirements established by the model can be traced through the structural design, functionality, and output(s) of the model; and secondly, to build confidence in the modeled results by attempting to prove that

the simulation model is, in effect, incorrect relative to the real-world system being modeled.

Finally, the V&V process allows to make an informed accreditation decision. Accreditation refers to the decision by a model sponsor (or its accreditation authority) whether to use a specific model for a specific application. Without an established and documented V&V process taking an appropriate decision will be almost impossible due to the lack of the critical knowledge to certify the model for its intended use. Bearing in mind the characteristics and needs to be covered in the Amalthea4public project, accreditation issues shall be not covered.

In the next figure a comparison between verification and validation is done.

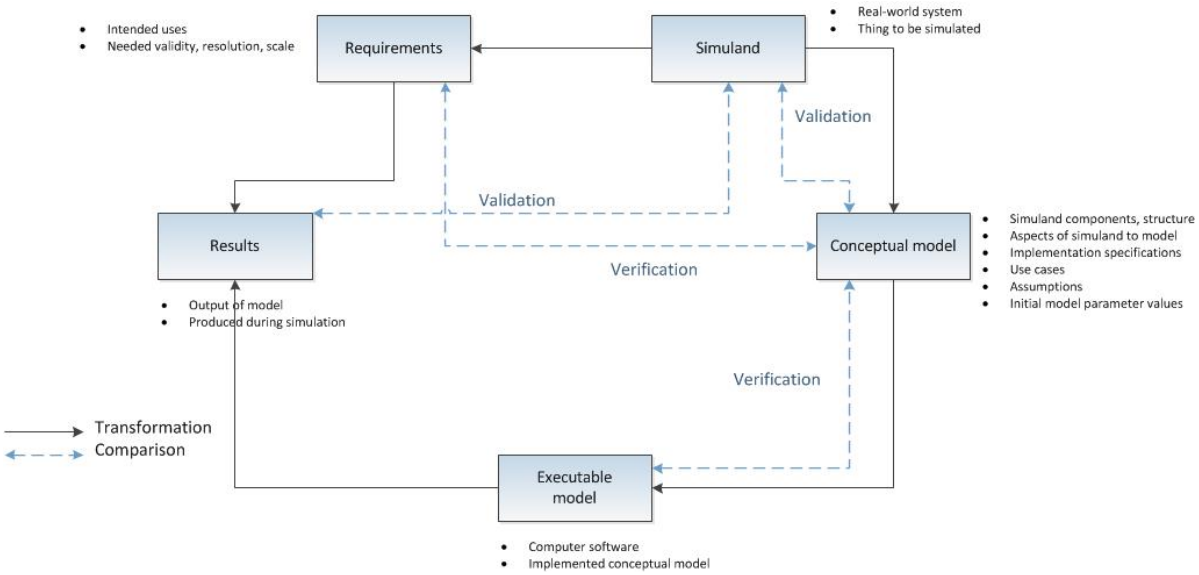

Figure 2.1: Verification and validation comparisons [89]

Verification can be made for a conceptual model against the requirements and the executable model, whereas the validation will be used to compare the real-world system/element to be simulated against its conceptual model and the results obtained as outputs produced during the simulations.

Developing an M&S application is not significantly different from any other software application regarding V&V considerations. However, certain types of M&S applications have more stringent V&V needs. M&S is typically used for one of three purposes: descriptive, predictive, and normative models.

- Descriptive models are intended to provide a characterization of the nature and workings of the modeled process to explain how a real-world activity functions.
- Predictive models, usually more complex than descriptive models, are designed to predict future events in addition to describing objectives and events.
- Normative (or control) models are the most difficult and complex models to construct since these models not only describe and predict, but also provide direction about the proper course of action.

Descriptive models require V&V just like any other software activity. Typical software has an output that, once verified and validated, can be used. Predictive and normative models

require additional V&V because the output of these models is used to predict and guide future actions, often without a human in the loop. Because of the potentially high cost of failure, V&V is critical for these types of models. Because of this, separate and additional V&V for the M&S is frequently merited. It is no secret that requirements are an integral part of all software activities. In fact, requirements engineering is fundamental to developing useful and valid software. Nowhere are requirements more important than in M&S. Prior to attempting to use M&S to help save time or costs in your system, make sure that a mature requirements engineering program is in place [107].

The are plenty of V&V available methods available for use since 1985 which address different purposes and they can be grouped in a variety of ways. The following categorization has been chosen due to its clearness distribution of methods inside groups (Informal, Static, Dynamic, Formal). The methods are related to each other depending on the similarities they present regarding the context they can be used and the differences between them concern the item which is being compared or the degree of formality and quantitativeness required.

The selection of V&V methods to be applied shall be done bearing in mind the characteristics of the model, the fact that certain methods apply best to specific types of model, or the type of data to be used because some methods will require specific types or amounts of data. In addition to this, other considerations are that some methods will need more resources or specific skills than others.

In the next table a summary of the most common V&V methods is presented. These techniques can be used in support of one or more of the V&V activities identified for the Amalthea4public project. Typically, these activities and techniques are carefully planned and documented in the form of a V&V Plan, which is revised as the V&V activity it describes unfolds during the course of the model development life cycle. DMSO provides this list as a set of tools from which a practitioner can select the most appropriate techniques for their particular project. It is not necessary, or even advisable to attempt to apply all of the techniques to any individual project. Many of the techniques are overlapping in their coverage, and it requires experience to determine which technique is the best to meet a project's needs.

Category	V&V techniques
`Informal`	• Audit • Face Validation • Reviews • Walkthroughs • Desk Checking • Inspections • Turing Test

Category	V&V techniques
Static	• Cause-Effect Graphing • Data Analysis – Data dependency – Data flow • Interface Analysis – Model interface – User interface • Structural Analysis • Syntax Analysis • Control Analysis – Calling structure – Concurrent process – Control flow – state transition • Fault/Failure Analysis • Semantic Analysis • Symbolic Evaluation • Traceability Assessment

Category	V&V techniques

Dynamic	• Acceptance Testing
	• Assertion Checking
	• Bottom-Up Testing
	• Compliance Testing
	  – Authorization
	  – Performance
	  – Security
	  – Standards
	• Execution Testing
	  – Monitoring
	  – Profiling
	  – Tracing
	• Field Testing
	• Graphical Comparisons
	• Object-Flow Testing
	• Predictive Validation
	• Regression Testing
	• Statistical Techniques
	• Structural (White-Box) Testing
	  – Branch
	  – Condition
	  – Data flow
	  – Loop
	  – Path
	  – Statement
	• Symbolic Debugging
	• Alpha Testing
	• Beta Testing
	• Comparison Testing
	• Debugging

Category	V&V techniques

Dynamic	• Fault/Failure Insertion Testing
	• Functional (Black-Box) Testing
	• Interface Testing
	  – Data
	  – model
	  – User
	• Partition Testing
	• Product Testing
	• Sensitivity Analysis
	• Special Input Testing
	  – Boundary value
	  – Equivalence partitioning
	  – Extreme input
	  – Invalid input
	  – Real-time input
	  – Self-driven input
	  – Stress
	  – Trace-driven input
	• Sub-model/Module Testing
	• Top-Down Testing
	• Visualization/Animation

Category	V&V techniques
Formal	• Induction • Logical Deduction • Lambda Calculus • Predicate Transformations • Inference • Inductive Assertions • Predicate Calculus • Proof of Correctness

Table 2.1: Overview of V&V techniques by category. Source: Verification, Validation, and Accreditation Recommended Practices Guide (DMSO, 1996)

Furthermore, the Figure 2.2 summarizes a number of characteristics of the general nature of each category: the basis for verification, the relative level of mathematical formality, the complexity of the associated techniques, cost in terms of human time and effort, cost with respect to computer resources (e.g., execution time, memory utilization, storage requirements, etc.), the relative effectiveness of the method in general, whether or not the category is considered instrumentation-based, and the relative importance of the associated techniques). the comparison among the categories is intended more to give a relative view among the spectrum of categories rather than to measure against some known standard.

In the subsequent sections further information will be provided regarding the categories identified for the distribution of the above listed techniques. Some techniques will be also explained with some detailed due to they could be interesting for the context of Amalthea4public project.

2.2 Informal methods

Informal methods are those which rely heavily on Subject Matter Expert (SME) expertise and evaluation. They usually are qualitative and subjective and performed by SMEs. The informal techniques have the advantage that they are relatively easy to perform and understand. However, it is commonly believed that these methods are unstructured. In fact, several of the methods such as desk checking or self-inspection can have very detailed checklists. Informal V&V techniques can be very effective if applied with structure and guidelines, and they are relatively low cost. These methods are effective for examining both the model and the simulation.

Some of the most used techniques of Informal V&V methods are explained next.

	Informal Analysis	Static Analysis	Dynamic Analysis	Formal Analysis
Category Definition	Analyzing through the employment of the informal design and development activities	Analyzing characteristics of the static source code	Analyzing results gathered during model execution	Formal mathematical proof of correctness
Level of Formality	Very Informal	Informal to Formal	Informal to Formal	Very Formal
Complexity	Low	Moderate	Moderate to High	Very High
Human Resource	Very High	Low to Moderate	Moderate to High	Very High
Computer Resource Cost	Very Low	Moderate to High	Very High	Very Low
Effectiveness	Limited	Moderate to High	Moderate to High	Highest, if Attainable
Instrumentation Based	No	No	Yes	No
Importance to PMV	High	High	High	Highest, if Attainable

Figure 2.2: Characteristics of the Programmed Model Verification techniques under each category

2.2.1 Verification

Audit

An audit is a verification technique performed throughout the development life cycle of a new model or simulation or during modification made to legacy models and simulations. An audit is a staff function that serves as the "eyes and ears of management" [87]. An audit is undertaken to assess how adequately a model or simulation is used with respect to established plans, policies, procedures, standards, and guidelines. Auditing is carried out by holding meetings and conducting observations and examinations [66]. The process of documenting and retaining sufficient evidence about the substantiation of accuracy is called an audit trail [88]. Auditing can be used to establish traceability within the simulation. When an error is identified, it should be traceable to its source via its audit trail.

Inspection

It is a general software verification method where requirements are compared to conceptual models and these conceptual models to executable models.

Teams of developers, testers and users will be organised to examine the product of a particular simulation development phase (e.g., M&S requirements definition, conceptual model development, M&S design).

The team inspecting a simulation design might include a moderator, a recorder, a reader from the simulation design team who will explain the design process and answer questions about the design, a representative of the Developer who will be translating the design into an executable form, SMEs familiar with the requirements of the application, and the V&V Agent. Normally, an inspection consists of five phases: overview, preparation, inspection, rework, and follow-up [96].

- Overview: The simulation design team prepares a synopsis of the design. This and related documentation (e.g., problem definition and objectives, M&S requirements, inspection agenda) is distributed to all members of the inspection team.
- Preparation: The inspection team members individually review all the documentation provided. The success of the inspection rests heavily on the conscientiousness of the team members in their preparation.
- Inspection: The moderator plans and chairs the inspection meeting. The reader presents the product and leads the team through the inspection process. The inspection team can be aided during the fault finding process by a checklist of queries. The objective is to identify problems, not to correct them. At the end of the inspection the recorder prepares a report of the problems detected and submits it to the design team.
- Rework: The design team addresses each problem identified in the report, documenting all responses and corrections.
- Follow-up: The moderator ensures that all faults and problems have been resolved satisfactorily. All changes should be examined carefully to ensure that no new problems have been introduced as a result of a correction.

Review

A review is intended to evaluate the simulation in light of development standards, guidelines, and specifications and to provide management, such as the User or M&S PM, with evidence that the simulation development process is being carried out according to the stated objectives. A review is similar to an inspection or walkthrough, except that the review team also includes management. As such, it is considered a higher-level technique than inspection or walkthrough.

A review team is generally comprised of management-level representatives of the User and M&S PM. Review agendas should focus less on technical issues and more on oversight than an inspection. The purpose is to evaluate the model or simulation relative to specifications and standards, recording defects and deficiencies. The V&V Agent should gather and distribute the documentation to all team members for examination before the review. The V&V Agent should also prepare a set of indicators to measure such as those listed in the table below.

The V&V Agent may also prepare a checklist to help the team focus on the key points. The result of the review should be a document recording the events of the meeting, deficiencies identified, and review team recommendations. Appropriate actions should then be taken to correct any deficiencies and address all recommendations.

Review indicators
Appropriateness of the problem definition and M&S requirements
Adequacy of all underlying assumptions
Adherence to standards
Modeling methodology
Quality of simulation representations
Model structure
Model consistency
Model completeness
documentation

Table 2.2: Review indicators

Walkthroughs

The main thrust of the walkthrough is to detect and document fault. It is not a performance appraisal of the Developer. This point must be made to everyone involved so that full cooperation is achieved in discovering errors. A typical structured walkthrough team consists of:

- Coordinator, often the V&V Agent, who organizes, moderates, and follows up the walkthrough activities
- Presenter, usually the Developer
- Recorder
- Maintenance oracle, who focuses on long-term implications
- Standards bearer, who assesses adherence to standards
- Accreditation Agent, who reflects the needs and concerns of the User
- Additional reviewers such as the M&S PM and auditors

Except for the Developer, none of the team members should be involved directly in the development effort. [18] [45] [81] [82] [113]

Inspections differ significantly from walkthroughs. An inspection is a five-step, formalized process. The inspection team uses the checklist approach for uncovering errors. A walkthrough is less formal, has fewer steps, and does not use a checklist to guide or a written report to document the team's work. Although the inspection process takes much longer than a walkthrough, the extra time is justified because an inspection is extremely effective for detecting faults early in the development process when they are easiest and least costly to correct [17] [31] [47] [74] [88] [96].

Inspections and walkthroughs concentrate on assessing correctness. Reviews seek to ascertain that tolerable levels of quality are being attained. The review team is more concerned with design deficiencies and deviations from the conceptual model and M&S requirements than it is with the intricate line-by-line details of the implementation. The focus of a review is not on discovering technical flaws but on ensuring that the design and development fully and accurately address the needs of the application. For this reason, the review process is effective early on during requirements verification and conceptual model validation. [66] [88] [100] [110]

Desk Checking / Self-inspection

Desk checking, or self-inspection, is an intense examination of a working product or document to ensure its correctness, completeness, consistency, and clarity. It is particularly useful during requirements verification, design verification, and code verification. Desk checking can involve a number of different tasks, such as those listed in the table below [31]. To be effective, desk checking should be conducted carefully and thoroughly, preferably by someone not involved in the actual development of the product or document, because it is usually difficult to see one's own errors [18].

Typical Desk Checking Activities
Syntax review
Cross-reference examination
Convention violation assessment
Detailed comparison to specifications
Code reading
Control flowgraph analysis
Path sensitizing

Table 2.3: Typical Desk Checking Activities

2.2.2 Validation

Face validation

With this validation method a comparison between the results and the simulation of the element is done. It implies doing an assessment based on expertise, estimates and intuition where the model validity is evaluated subjectively. This method is frequently used because its simplicity and when user interaction is important.

The project team members, potential users of the model, and subject matter experts (SMEs) review simulation output (e.g., numerical results, animations, etc.) for reasonableness. They use their estimates and intuition to compare model and system behaviours subjectively under identical input conditions and judge whether the model and its results are reasonable [64].

Face validation was used in the development of a simulation of the U.S. Air Force (AF) manpower and personnel system to ensure it provided an adequate representation. The simulation was designed to provide AF policy analysts with a system-wide view of the effects of various proposed personnel policies. The simulation was executed under the baseline personnel policy and results shown to AF analysts and decision-makers who subsequently identified some discrepancies between the simulation results and perceived system behaviour. Corrections were

made and additional face validation evaluations were conducted until the simulation appeared to closely approximate current AF policy. The face validation exercise both demonstrated the validity of the simulation and improved its perceived credibility.

Face validation is regularly cited in V&V efforts within the Department of Defense (DoD) M&S community. However, the term is commonly misused as a more general term and misapplied to other techniques involving visual reviews (e.g., inspection, desk check, review). Face validation is useful mostly as a preliminary approach to validation in the early stages of development. When a model is not mature or lacks a well-documented V&V history, additional validation techniques may be required.

Turing test

A comparison is made between model behaviour and human behaviour. This validation technique is suitable for human behaviour models and when it is not possible to make distinctions between them it can be concluded that the model-generated behaviour is valid or realistic.

System experts are presented with two blind sets of output data, one obtained from the model representing the system and one from the system, created under the same input conditions and are asked to differentiate between the two. If they cannot differentiate between the two, confidence in the model's validity is increased [99] [105] [68]. If they can differentiate between them, they are asked to describe the differences. Their responses provide valuable feedback regarding the accuracy and appropriateness of the system representation.

2.2.3 Advantages and disadvantages of Informal Analysis

Informal analysis can be of great importance. Its techniques are valuable from the early stages of model formulation throughout the entire programming process. In particular is the ability of informal analysis techniques to evaluate the subjective and multifaceted aspects of the simulation study. The success of a simulation study stems from the ability to achieve sufficiently correct simulation results and as importantly, to convince the study sponsor that the simulation model is a sufficiently accurate one. Insuring the acceptance of the many subjective aspects of the model cannot be overlooked. Besides the advantage of allowing human reasoning in the verification process, the informal analysis techniques are not difficult to perform and require virtually no computer resources. On the other hand, the techniques used are very time consuming and require very high human resource allocation. Because of their reliance on human evaluation they are prone to human error. Success depend on the level of knowledge and expertise of the individual. The human time and effort required coupled with the likelihood of error result in limited effectiveness of informal analysis. Though their effectiveness improves as their guidelines for use become more structured and formal, informal analysis techniques cannot be relied upon in themselves to verify the programmed model.

2.3 Static methods

The static V&V methods are based on artifact characteristics that can be determined without running a simulation. They often involve analysis of executable model code and may be supported by automated tools or manual notations or diagrams. They are more often performed by technical experts. These techniques are used almost exclusively to examine the model and its implementation. Unfortunately, static V&V techniques do not examine the execution of the

model, and therefore they are of limited usefulness in M&S V&V. Static V&V techniques can be used on the code of the model as it is being developed, but they are not effective on simulations themselves, as simulations require execution of the model. They only perform V&V on the model, and ignore the simulation. Because of that, during M&S V&V, dynamic techniques are also used for simulation.

These techniques can obtain a variety of information about the structure of the model, coding techniques and practices employed, data and control flow within the model, syntactical accuracy, and internal as well as global consistency and completeness of implementation. The information gathered can be used to generate test data for use with other types of analysis, can identify the testing requirements for the various areas of the model, can be used to optimize the model's code, and can even be used to instrument the model to enhance further analysis. Just as importantly, their results provide an indication of the principles used to meet the objectives of the software development project [23]. Knowing that the model is being engineered for quality makes a strong statement for verification.

Static analysis is generally more complex than informal analysis but not as complex as the other categories of analysis.

Examples of static methods are shown next.

2.3.1 Verification

Syntax analysis

Any model that is to undergo translation from a higher form to a machine-readable form must first pass a syntax check. This check assures that the mechanisms of the language are being applied correctly. This fundamental analysis is the most widely utilized verification technique.

During the course of a compilation, as the syntax is checked and the source statements 'tokenized', a symbol table is built which describes in detail the elements, or symbols, which are being manipulated in the model. This includes descriptions of all functions declarations, type and variable declarations, scoping relationships, interfaces, dependencies, and so on. The symbol table holds the compilation together, growing dynamically as the source code is scanned. Obviously there is a wealth of information about the static model available in the symbol table. Just listing the table itself is a tremendous source of documentation.

In addition to this, cross-reference tables are easily generated which provide such information as called versus calling submodels, where each data element is declared, referenced and altered, duplicate data declarations and unreferenced source code. Submodel interface tables reflect the actual interfaces of the caller and the called, particularly useful when using a compiler that does not perform strict type checking nor verify external calls. Also readily created are maps which relate the generated runtime code to the original source code. All of this information is useful for documentation purposes. It is even more useful as the underpinnings for debugging. Another useful feature is the ability to reformat the source listing on the basis of its syntax and semantics.

All of the above have various merits for documentation and display of the source model, and even the model specifications.

Semantic analysis

Also occurring during source code translation is semantic analysis. It attempts to determine the modeller's intent in writing the code. The goal is to obtain an accurate translation of

modeller's intentions. In truth, the only meaning which can be derived from the source code is that which is self-evident in the code. It is dangerous to let the compiler make any other assumptions about modeller's intentions. It therefore becomes beneficial, even to the point of being essential, to tell the modeller what it is that he has specified in the source code. The same principle can be applied to specifications. It is then up to the modeller to verify that the true intent is being reflected.

When the source code is being parsed during compilation, the target runtime system is most likely being simulated. This allows the compiler to generate code which will perform the requested tasks. As the meaning of the source code is derived, the corresponding runtime code is produced. The symbol table is referenced to check that the data elements used fit the operation being performed. A result of this is the ability to determine what is and is not being used, how often it is being used, and to a large degree in what manner it is being used. As in syntax analysis, the harnessing of this information provides a healthy source of documentation.

Other benefits include locating variables which have been used but not initialized.

Neither syntax analysis nor semantic analysis require complete compilation in order to obtain their results. Like the results of syntax analysis, semantic analysis should be captures and maintained to drive other parts of the verification process. The usefulness of this data will become self-evident as dynamic analysis techniques.

Structural analysis

Structured design and development refers to the use of widely accepted techniques for constructing quality software. These techniques are all founded on a set of principled which are recognized to be effective and comprehensive building blocks for software development. The principles are based on the use of acceptable 'control structures' from which the software will be built. The three basic control structures are sequence, selection and iteration.

Structural analysis examines the model's structure and determines if it adheres to structured principles. This is accomplished by constructing a graph of the model control structure. This graph defines model control flow and as such is called a control flow graph. The control flow graph is analysed for anomalies, such as multiple entry and exit points, excessive levels of nesting within a structure, and questionable practices such as the use of unconditional branches. The anomalies can be flagged so that they may be scrutinized further.

Structural analysis may also reveal commonalities of particular model structures. Steps may be taken to reduce the structure if possible.

The control flow graph is an effective verification document due to it documents the model's control flow in a clear an concise way.

Data flow analysis

Data flow analysis is concerned with the behaviour of the programmed model with respect to its use of model variables. This behaviour is classified according to the definition, referencing, and unreferencing of variables [18], i.e., when a variable space is allocated, accessed, and deallocated. A data flow graph can be constructed to aid in the data flow analysis. The nodes of the graph represent statements and corresponding variables. The edges represent control flow.

Data flow analysis can be used to detect undefined or unreferenced variables (much as in static analysis) and, when aided by model instrumentation, can track minimum and maximum variable values, data dependencies, and data transformations during model execution. It is also

useful in detecting inconsistencies in data structure declaration and improper linkages among submodels [92].

Consistency checking

Consistency checking is essential to the integrity of the model. It is concerned with verifying that the model description does not contain contradictions. All specifications must be clear and unambiguous so that each person viewing the model sees the same thing. All model components must fit together properly. It also involves verifying that the data elements are being manipulated properly. This includes data assignment to variables, data use within computations, data passing among submodels, and even data representation and use during model input and output. Must consistency checking is accomplished by using the documentation produced by syntax and semantic analysis (listings, cross-references) as materialto guide code inspections and walkthroughs. As the specification becomes more formally stated, more of the work can be automated. Data elements and interfaces can be checked as they are actually used to ensure their consistent usage.

Another perspective on consistency checking is related to the cosmetic style with which language elements are applied (e.g., naming conventions, use of upper, lower, and mixed case, etc.).

2.3.2 Validation

Cause-effect graphing

This validation technique compares causes and effects in the element to be simulated to those in the conceptual model. Cause is understood as the event or condition whereas the effect is the state change triggered by a cause. With this method it is possible to identify missing, extraneous, and inconsistent cause-effect relationships.

The technique assists accuracy assessment by addressing the question of 'what causes what in the model representation?'. It is performed by first identifying causes and effects in the problem domain being represented and by examining if they are accurately reflected in the model specification. As many causes and effects as possible are listed and the semantics are expressed in a cause-effect graph. The graph is annotated to describe special conditions or impossible situations. Once the cause-effect graph has been constructed, a decision table is created by tracing back through the graph to determine combinations of causes which result in each effect. Finally, the decision table is then converted into test cases with which the model is tested.

2.3.3 Advantages and disadvantages of Static analysis

Most static analysis techniques have automated tools which support their use. As a result, the human resource cost is appreciably low. Since model execution is not involved, computer resource cost is moderate compared to instrumentation-based verification approaches. These techniques are limited, however, in what they can actually verify. For instance, static analysis can verify that the syntax used conforms to the defined syntax of the language. It can make conclusions about the semantics of the model and inferences on aspects of the model's execution. It cannot insure that the intentions of the modeller are being met nor can it algo-

rithmically examine a model to determine its execution behaviour [52] [67]. Further, the basis for performing the verification must be shown to be correct (e.g., the compiler must be correct).

Overall, static analysis has proven to be an effective verification method. Its strength lies in the number of well-known techniques which are supported by a variety of commercial available tools, most of which are highly automated. Further, static analysis complements other methods of verification, such as symbolic execution and execution profiling.

Especially important to the simulation study is the extensive documentation generated through static analysis. Graphs which depict the model's logic and data flow are easily understood even through the layman's eyes. The construction of the model can be shown to be structurally sound and free of any anomalies which might arouse questions about the model's integrity.

2.4 Dynamic methods

Dynamic methods are those methods that involve running the executable model and assessing the results. The results can be compared due to they often are quantitative and objective as well as performed by technical experts. Dynamic V&V techniques look at the results of the execution of the model. At the simplest level, dynamic V&V can be merely examining the output of an execution. However, that is almost always insufficient. Instead, the model must be examined as it is being executed. This typically requires instrumenting the insertion of additional code into the model to collect or monitor model behaviour during execution. Normally, the steps involved are to instrument the model with V&V code, execute the model, and then analyse the dynamic behaviour and output of the model. While these are extremely useful techniques, instrumenting a model changes it slightly. To observe the dynamic execution of a model requires additional instructions to collect data. These additional instructions can slightly modify the timing or behaviour of the model. A dictum to remember in dynamic V&V is, 'Those who observe, perturb!'. Because of that great care must be used in instrumenting simulation code to ensure that the instrumentation itself does not affect the validity of the simulation output.

Examples of dynamic methods presented below.

2.4.1 Verification

Black-box testing

Black-box testing is concerned with what the model or submodel does, i.e, what its function is. Black-box testing (also known as functional testing) views the model (the "test object") as a black box. The concern is not what is in the box; rather, what is produced by the box. Testing of the model is accomplished by feeding inputs to the model and verifying the corresponding outputs. The model specification is used to derive tests data [82] [69].

In general it is virtually impossible to test all inputs to the model. Rather than verifying that the model produces the correct output for each input, the modeler is more interested in finding inputs that produce incorrect outputs. Determining if the test set is complete is the main drawback to black-box testing [108]. Black-box testing is typically used at the global model level, when all of the submodels have been thoroughly tested with another approach.

White-box testing

As opposed to black-box testing, which tests the function of a model, white-box testing tests the model based on its internal structure (how it was built). White-box testing uses data flow and control flow graphs to verify the logic and data representations of the model. The focus of testing here is breadth of coverage of model paths. As many execution paths as possible should be tested.

White-box testing is the most common mode of testing. It is the only reliable means of detecting redundant code, faulty model structure, and special case errors [108]. An effective test plan determines which approach best fits the varied needs of the model and applies them accordingly. In most cases, all approaches will be used in some way, blended together in a well-orchestrated, concerted manner.

Stress testing

A characteristic of simulation software is a dependency on time. Quite often real-time requirements and tight synchronization are involved. Testing these time-dependent situations is a difficult task. Many testing techniques are not adequate for these particular needs.

An approach to time-sensitive testing needs is stress testing. Stress testing tests the model on the borders of its time critical components. It pushes the model to and beyond its limits. If the model performs well under both valid and invalid input conditions, the model is said to be robust. As [82] points out, such tests are valuable because such "never-will-occur" situations may, in reality, occur, and system response under such conditions is often indicative of errors that might occur under "normal", less stressful conditions.

Stress testing, while in no way considered an exhaustive testing technique, is valuable for giving evidence (along the lines of strength in numbers) that a model will behave as desired if, after numerous stressful tests have been performed, no errors arise. Lack of errors do not imply correctness; however, stress testing provides an alternative to not having any functional evidence at all. It is important that any test plan involving stress testing be strongly supported with a solid structural testing program.

Execution tracing

The method can be used in verification and validation contexts and it consists on recording and examining the simulation executions. It can be done 'line by line' or 'step by step'. The output simulation state variables change at each state and state examinations must be done for consistency and reasonableness. Comparisons of results to conceptual models and the elements of simulation are done. The outputs generated can be to GUI or trace file and the examination manual or automated.

The modeller can view the model's execution, determine what factors cause the traversal of particular paths, follow model data flow, determine in what order data elements combine and how the data is treated, and so on. Tracing is like creating a window into the execution environment. The modeller can see what is happening at specific locations in the model, recreate the events of the simulation, and easily track the source of errors.

Execution tracing is most often associated with interpretive languages which offer source level tracing by simply displaying the source statement being interpreted at the given moment. The tracing features and closeness to the source code of interpretive languages make this an

attractive alternative. And languages with trace capability provide a mechanism for turning tracing on and off.

Trace data can be displayed during execution or routed elsewhere for subsequent analysis and use. [50] [51] suggests maintaining the trace data in a database in order to enhance further verification activity.

Although execution tracing can be used to verify the model, other techniques are often easier to use, with the same or greater effectiveness. Typically, tracing is used to aid debugging by isolating known errors in the code.

This technique can be also used in terms of verification.

Regression testing

As model development progresses the model is going to evolve: incorporate design changes and correct mistakes. Verification is also a continuous process, flowing with the tide of change.

When mistakes are corrected, the corrections often result in adverse side-effects to the existing model. If care is not taken, the correction of an error in one place leads to an error in another. The later in the life cycle error correction takes place, the greater the likelihood of harmful side-effects occurring. Regression testing seeks to assure that model corrections do not initiate other problems. Regression testing is usually accomplished by retesting the corrected model with a subset of the previous test sets used. This makes retaining and managing old test data essential. Successful regression testing is as much a matter of planning and configuration control (simulation project library management, version control, traceability, etc.) as it is anything else. Thus a plan for performing regression testing must be incorporated in the overall model design. Waiting until the first (sub)models begin undergoing correction and revision is too late to think about regression testing.

Comparison testing

Simulations and their corresponding scenarios are run using two different models. Afterwards, the results are compared. When differences between results are found then further research is needed because it can be a signal of problems. The key is to find which model has problems in case of differences, so depending on the assumptions made, this method will be performed as verification or validation:

- One model is assumed as valid –> validation method
- Neither model is assumed as valid –> verification method

This technique can be also used in terms of verification.

2.4.2 Validation

Statistical methods

Statistical validation methods are used to compare model results to the observations made to the real-system or the elements under simulation. Various statistical methods can be used, sometimes combined with other methods: regression analysis, analysis of variance, confidence intervals, hypothesis tests, etc. Each statistical method defines statistic or metric of 'closeness' or similarity, it is a measure of validity. However it is a method that is generally underutilized. Examples of statistical methods are shown in the next table.

Model(s)	Statistical method	Reason for selection
Spacecraft propulsion system sizing tool	Regression	Paired data Simuland model
Medical clinic waiting	Confidence intervals	Single simuland observation Multiple model runs
Seaport loading/unloading		
Historical tank battle		
Bombing accuracy MC	Confidence intervals with error tolerance	Single simuland observation Multiple model runs Error tolerance available
Bank drive-up waiting line	Hypothesis test comparing distributions	Multiple simuland observations Multiple model runs
Entity-level combat		
Command decision making	Hypothesis test for equivalence	Multiple simuland observations Multiple model runs Assumption of equality avoided
Missile impact MC	Hypothesis test comparing variances	Multiple simuland observations Multiple model runs

Figure 2.3: Application of statistical methods

2.4.3 Advantages and disadvantages of Dynamic analysis

The potential cost in human resources for dynamic analysis can be very high. If not managed properly, dynamic analysis can needlessly consume the time of the modeller. Secondly, dynamic analysis cannot show model correctness. It can only reflect how the model behaves for a given set of test data. The possible test sets for a model can be infinite. Thus complete testing is rendering impossible for virtually all practical models of any speakable size. Adequate test coverage is a problem as well. The required scope of coverage broadens in exponential fashion as the model increases as the model increases in size. Dynamic analysis does not possess the capability to manage this situation.

On the other hand, dynamic analysis techniques thoroughly document a given test execution. It can provide conclusive proof that a model functioned as intended. dynamically executing the model is the only way to test how the model behaves on a given hardware, or when operating on distributed hardware. The execution history not only enhances error detection and correction, it serves as a reference of model structure which can be used to enhance error detection and correction, it serves as a reference of model structure which can be used to enhance and maintain the model. Combining dynamic analysis with other verification techniques helps reduce some of the problems associated with dynamic analysis.

2.5 Formal methods

The formal methods are based on formal mathematical proofs of program correctness. They are quantitative (or logical) and objective. If attainable, formal proof of correctness is the most effective means of verifying software. The term 'correct' means that the model meets its specifications. Therefore, formal proof of correctness corresponds to expressing the model in a precise notation and then mathematically proving that the executed model terminates and that it satisfies the requirements of its specification.

Traditionally, natural language is used to specify the artefacts and products in the software development. Often specifications depend on the semantics/meanings of words to convey the understanding to the system to be developed. Due to the ambiguity of the natural language, it is difficult to verify the correctness of the system and the verification process becomes less effective and less rigorous when the computer system becomes larger and more complex. The automation of the process is also impossible because of the informal specifications. Thus, the needs for formal methods increase tremendously especially in the fields of distributed system, concurrent system, and real-time system development. The uses of formal methods enhance the understanding of software requirements. Formal methods also enable rigorous verification of specifications and their implementation. They facilitate the automation of the verification process and improve the effectiveness of the verification process. Thus, the use of formal methods is expected to lead to increased software quality and reliability.

The growing complexity of designs increases the importance of verification techniques. The research in the formal verification of hardware and software has lately made significant progress in developing methodologies and tools.

Formal verification is the process of determining whether or not the products of a given phase in the life-cycle fulfil a set of established requirements, using a formal mathematical notation. During software development, the code must not only implement behaviours as specified by a model, but a model itself may need to change based on discovered limitations of the implementation environment or changes of customer requirements. To reduce the divergence between the code and models which may cause future problems such as design errors, expensive rework, etc., the formal verification has to be conducted throughout the whole development cycle. So Formal verification mathematically prove the correctness of a design with respect to a mathematical formal specification

Formal verification conducts exhaustive exploration of all possible behaviours and are performed by technical experts. In the following table a comparison between formal verification and testing is provided.

Formal design validation combines aspects of traditional checking and dynamic simulation based verification with the symbolic simulation and static analysis techniques of formal verification to provide optimized trade-offs in scalability and completeness so improving verification effectiveness. Formal validation checking tools extend the ease of use methodology of traditional checking techniques to the area of property checking and so provide a practical and usable alternative to difficult to use formal verification techniques, such as model checking.

Examples of formal methods are presented next.

2.5.1 Verification

Inductive assertions

This verification method aims to construct proof of executable model correctness. In order to achieve this goal assertions and statements about required executable model input-to-output relations are associated with execution paths in an executable model. Basically proofs of assertions along paths are constructed and those proofs along all paths imply correctness. The executable model is then compared to the conceptual model. This specific method is closely related to general program proving techniques, where proofs are done using mathematical induction and the 'Correctness' is with respect to the conceptual model. An example of inductive assertions methods are assertions concerning to the performance of a fiber optic system model,

fit actual performance parameters to the components of the system model, design the system model, check to see if the input signals produce the required output signals and adjust parameters to obtain the desired result.

Model Checking

With Model Checking methods (cf. [25]) a model can be formally verified with respect to specified properties. Model checking uses symbolic simulation techniques to verify that these properties are satisfied for all legal design inputs. In symbolic simulation, symbols are used rather than just logic zero and one values. Temporal logic (CTL - Computation Tree Logic, LTL - Linear temporal logic, etc.) allows the user to specify these properties. Then the model will be executed at a symbolic level in order to proof the correctness of the specified properties (see Figure 2.4). Symbolic model checking is primarily useful in verifying the control parts (i.e., the state based behavior) of a system. When applied naively it is impractical for most data paths, since it suffers from the state explosion problem where the state space of the design that must be explored grows extremely large (cf. section 3.4.6).

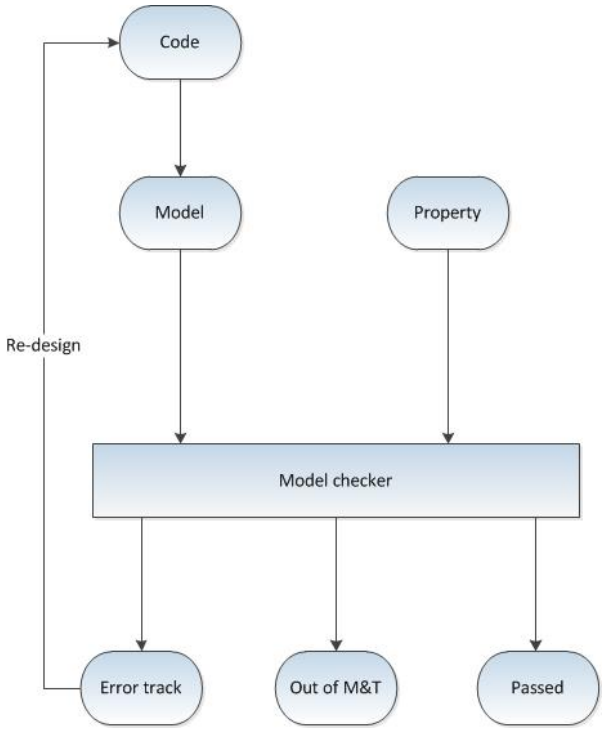

Figure 2.4: Model checking (verification)

In addition, performing model checking on design modules requires that an interface specification for the module is available so that only legal inputs are considered. However, in practice, detailed interface specifications of design modules are rarely available and so verification often requires the creation of the interface specification. This can be a complex task.

The model checking formal verification approach is heavily dependent on experienced users who must specify the properties of the design that are to be checked. The reliance on a design engineer, who must provide knowledge about design behavior and the design properties to be

analyzed, has limited the adoption of this technology.

Deductive Methods

A deductive verification is a formal verification method that uses an expressive logic to state correctness of a given target system relative to a given property. Logical reasoning (deduction) is used to prove validity of that property [29].

The first question is that of defining the correctness of a program. In other words, what is the program supposed to do and how can we express it formally? This is done using a specification language. This is made of two parts: a mathematical language to express the specification and its integration with a programming language. The use of a specification language is needed by deductive verification, but can also be used for documenting the program behaviour, for guiding the implementation, and for facilitating the agreement between teams of programmers in modular development of software. The specification language expresses preconditions, postconditions, invariants, assertions, and so on. This relies on Hoare's work [65] introducing the concept known today as Hoare triple.

This technique contrasts to model checking (propositional temporal logic specifications), correctness by refinement/sound compilation (preservation of behaviour) and abstract interpretation (abstraction of target system).

There are three main approaches:

- Proof Assistant: it is an interactive proof system for an expressive, higher-order logic. It is based on few axioms (e.g., ZF set theory [16]) and other rules soundly derived. It is encoded in syntax and semantics of target system as well as properties. It is independent of target language, very general and can be used to prove meta properties (e.g., compiler correctness). However, the coding effort for a new system is substantial and it requires a high degree of interaction and expertise.

- Program Logic: it uses a modal logic tailored to reason about a specific target language and involves calculus for that logic whose rules reflect semantics of target language. An interleave analysis of target system and intermediate simplification are performed. There is a high degree of automation achievable ($> 99\%$ of rule applications) and verification flow follows flow of execution of target system. On the other hand, the implementation effort for a new language is substantial and detailed auxiliary specifications may be required (e.g., invariants).

- Verification Condition Generation (VCG): this technique annotates target program with assertions that are propagated through it, which results first-order verification conditions discharged by SMT solver. It involves "Batch mode" interaction pattern (edit; run; analyse) and bug finding. As advantages the following can be mentioned: "run" part is fully automatic and loose coupling of VCG, SMT (making use of latest SMT solver technology). On the contrary, expressiveness of assertion language usually limited (e.g., quantifier-free) and analysis of a failed run hard to interpret.

2.5.2 Validation

Predicate calculus

It involves a method to logically analyse a conceptual model. It is a formal logic system to create, manipulate, and prove statements. The conceptual model is compared to the real-world system/the elements to be simulated and the conceptual model are described in predicate calculus, so it is used to prove properties of both to show logical consistence. However, it is quite difficult to apply to non-trivial problems.

The programmed model can be defined in terms of predicate and manipulated using the rules of the predicate calculus. The predicate calculus forms the basis of all formal specification languages. Predicate transformation provides a basis for verifying model correctness by formally defining the semantics of the model with a mapping which transforms model output states to all possible model input states. This representation provides the basis for proving whether or not the model is correct (if it has transformed initial states to termination states properly). Figure 2.5 shows an example of Predicate calculus.

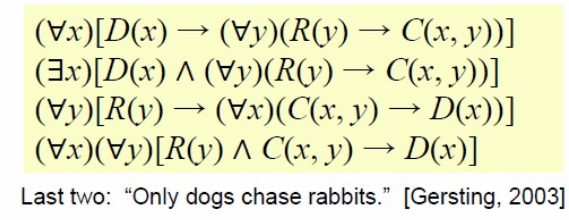

Figure 2.5: Example of Predicate calculus method for validation

2.5.3 Advantages and disadvantages of Formal analysis

Attaining proof of correctness in a realistic sense is not possible with current technology but is a field of active research. Setting up a proof for even a simple model is an expensive, time-consuming undertaking today. Completing the proof would be just as intense. The matter is further complicated by non-mathematical considerations such as machine dependencies and other related idiosyncrasies. However, the advantage of realizing proof of correctness is considerable, so when it is realized, it implies a big difference in the verification of software. For example, MechatronicUML aims at providing a holistic compositional verification approach to handle the state-space-explosion problem (cf. 3.4.6).

2.6 Verification using Test Cases

2.6.1 Test Cases generation

Model based testing (MBT) refers to the type of methods using model(s) as base element(s) for all testing activities. In general it is used as a technique for generating test cases (see Figure 2.6). Based on a specification model test cases wil be generated with respect to specified test goals (coverage criteria, test metrics, etc.). The specification model describes the expected behavior of the test object at an abstract level.

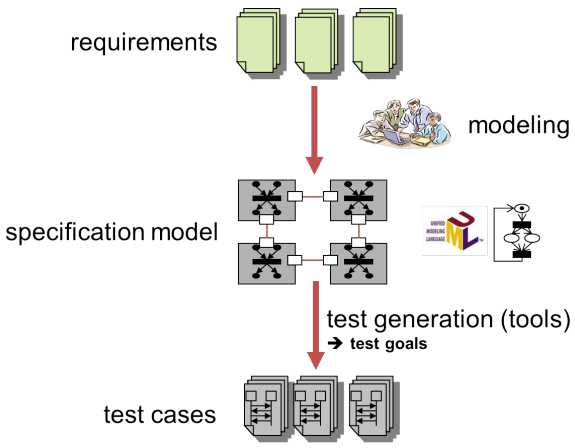

Figure 2.6: General process for model based test generation

But also other testing activities like test implementation, test execution, documentation, etc. can be simplified with the using of models. Those formal models are generally categorized into three main categories: requirements models, usage models, and source code dependant models. The requirements models can be behavioural, interactional, or structural models according to the perspective by which the requirements are being looked at. The test cases derived from behavioural or interactional models are in general functional test cases and they have the same level of abstraction as the models creating them. These kinds of test cases differ from those derived using structural models. Other types of models can be used as well to extract test cases; often the Unified Modeling Language (UML) is used for creating and describing specification models on order to generate test cases.

The more early test cases are generated, the more costs, time and effort can be saved when the actual testing time comes. Many researchers have recently given this field a great attention where test cases can be generated in the analysis and design phases using requirements-based models and sometimes other models. UML diagrams are the most common type of models used to represent the requirements-based models. They can be categorized into behavioural, interactional and structural diagrams.

Behavioural diagrams are a type of diagrams that represent behavioural features of a system or business process. They include activity, state chart, and use case diagrams as well as the four interaction diagrams (communication, interaction, sequence and timing).

Interactional diagrams are a subset of behavioural diagrams which accentuate object interactions. They include communication, interaction overview, sequence, and timing diagrams.

Structural diagrams are a type of diagrams that emphasize the elements of a specification which are irrespective of time. They include class, component, deployment, object, composite structure and package diagrams.

The categorization of UML diagrams yields to a categorization of the test cases generation techniques according to the diagram(s) being used. An extra category is given to generation techniques that use other types of models rather than the UML models like the mathematical, boolean and feature models. The categorized techniques are classified as follows.

Behavioral and Interactional UML Model-based Techniques Activity diagrams are graphical representations of workflows of stepwise activities and actions with support for choice,

iteration and concurrency. In the Unified Modeling Language, activity diagrams are intended to model both computational and organisational processes (i.e. workflows). Activity diagrams show the overall flow of control. Activity diagrams are constructed from a limited number of shapes, connected with arrows Figure 2.7.

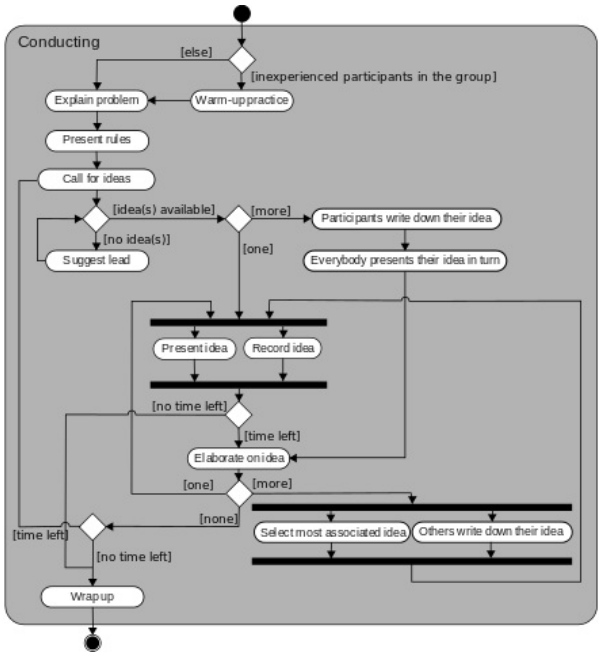

Figure 2.7: Activity diagram

The use of model checking and activity diagrams is the aid of the approach proposed in Coverage-driven Automatic Test Generation for UML Activity Diagrams [37]. The UML activity diagram is translated into a formal model which is considered the NUSMV input [9]. Next, properties in the form of CTL (Computational Tree Logic) or LTL (Linear Temporal Logic) formulas are generated using coverage criteria. Finally, the properties are applied on the NUSMV input using model checking to generate required tests.

Other types of diagrams have been used in many approaches to generate test cases like State chart, Collaboration, and Sequence diagrams.

State charts are used to give an abstract description of the behaviour of a system. This behaviour is analysed and represented in series of events, that could occur in one or more possible states. Hereby each diagram usually represents objects of a single class and tracks the different states of its objects through the system. State diagrams can be used to graphically represent finite state machines Figure 2.8.

An algorithm that transforms a state chart diagram into an intermediate diagram, called the Testing Flow Graph (TFG) is shown in [70]; from the TFG it generates test cases that apply the full state and full transition overage criteria.

Collaboration diagrams model the interactions between objects or parts in terms of sequenced messages. Collaboration diagrams represent a combination of information taken from Class, Sequence, and Use Case Diagrams describing both the static structure and dynamic behaviour of a system.

However, Collaboration diagrams use the free-form arrangement of objects and links as used

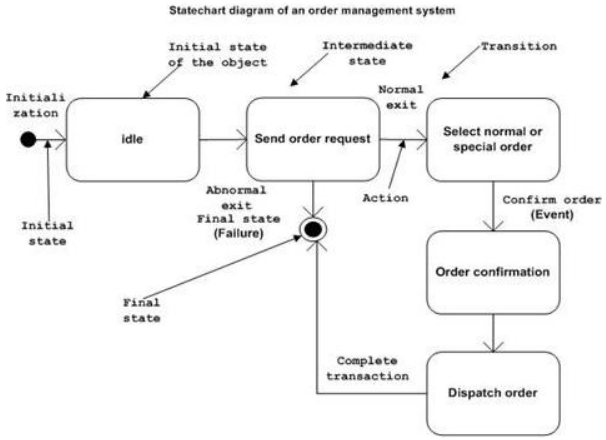

Figure 2.8: State chart diagram

in Object diagrams. In order to maintain the ordering of messages in such a free-form diagram, messages are labelled with a chronological number and placed near the link the message is sent over. Reading a communication diagram involves starting at message 1.0, and following the messages from object to object Figure 2.9.

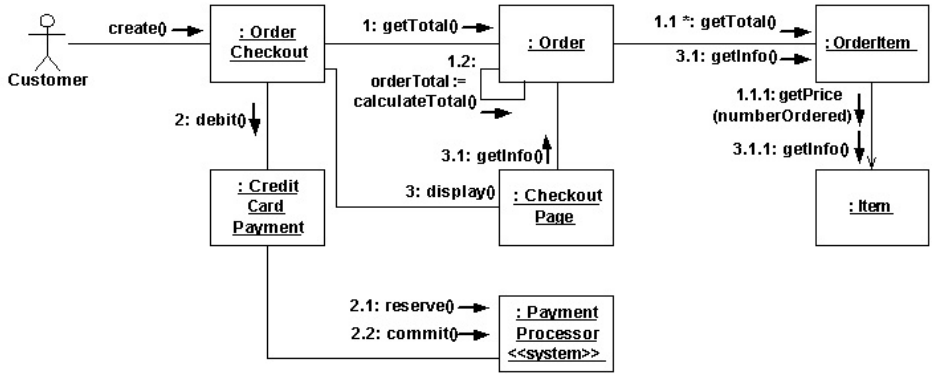

Figure 2.9: Collaboration diagram

Collaboration diagrams are represented using trees in the approach presented in automatic test case generation from UML communication diagrams [94]. The approach after constructing a tree out of the system's collaboration diagram carries out a post-order traversal on it for selecting conditional predicates. Then, it applies function minimization technique to generate test data. The generated test cases achieve message paths coverage as well as boundary coverage criteria.

Sequence diagrams are interaction diagrams that show how processes operate with one another and what is their order. They are constructs of Message Sequence Charts. A sequence diagram shows object interactions arranged in time sequence. It depicts the objects and classes involved in the scenario and the sequence of messages exchanged between the objects needed to carry out the functionality of the scenario. Sequence diagrams are typically associated with use case realizations in the Logical View of the system under development. Sequence diagrams are sometimes called event diagrams or event scenarios.

A sequence diagram shows, as parallel vertical lines (lifelines), different processes or objects that live simultaneously, and, as horizontal arrows, the messages exchanged between them, in the order in which they occur. This allows the specification of simple runtime scenarios in a graphical manner Figure 2.10.

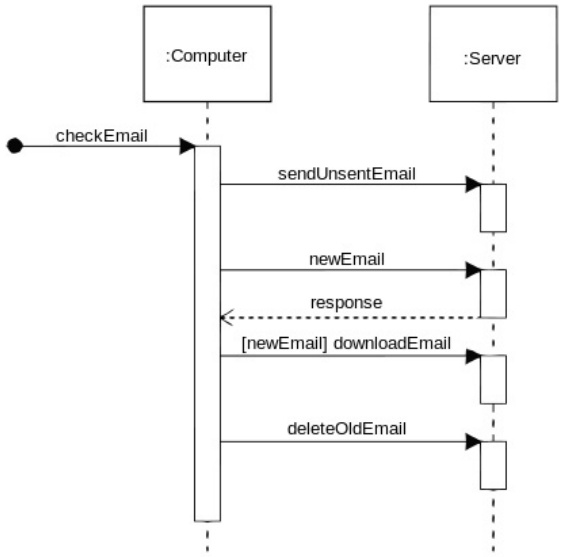

Figure 2.10: Sequence diagram of e-mail message sequence

UML sequence diagrams are also used to generate test cases. Transforming them into graphs called the sequence diagram graphs (SDGs) is the first step to do so [95]. The presented approach then increases the SDG nodes with different information necessary to form test vectors. The test vectors are finally reformed to represent the test cases.

Altering sequence diagrams to have an initial model and making this model the starting point of the algorithm is another way of generating test cases for unit testing. It is shown in [71]. The sequence diagram is first transformed into a general unit test case model called xUnit using model-to-model transformations. Then the general xUnit model is transformed into platform specific (JUnit, SUnit etc.) test cases using model-to-text transformations.

Sequence diagrams can be used with activity diagrams as well to generate test cases in a strategy shown in Test Case Generation from Behavioral UML Models [101] where one general sequence diagram is built for each use case. The constructed sequence diagram is then used to create several intermediate tables and flow graphs that are used in turn to create test sequences. The created test sequences are what this strategy uses to extract its final test cases.

Petri net based techniques of test case generation of concurrent behavior A similar approach to the UML Model-based techniques is the generation of test cases out of an Petri net as specification model. Thereby a Petri net can be seen as a generalization of a UML State Machine. There are a lot of mapping algorithms for transforming a UML State Machine into a suitable Petri net available. With a Petri net the modeling of concurrent behavior is quite simple and intuitive, which can be an advantage if modeling of requirements with concurrent behavior is necessary. The procedure of generating test cases based on a Petri net as specification model is similiar to the procedure for other graph based specification models like UML

State Machines, Statecharts, UML Activity diagrams, etc. The graph will be traversed and test cases will be derived with respect to specified test goals. Only for test goals with quite high coverage criteria of the specification models, like all path must be covered with at least one test case, the whole behavior of the specification model will be calculated. For Petri nets the whole behavior is represented by its reachability graph or by the complete prefix of the unfolding [73]. But this is often not possible because of the mentioned state explosion problem.

Requirement based techniques of test case generation The greatest effort within a model based test generation process is the creation of the specification model as basis of test generation algorithms. It is very difficult for test engineers to model all requirements, described with natural language within requirement documents, with one behavior model, e.g. with one UML State Machine, or with one UML Acitvity diagram, or with one Petri net. Therefore there are some academic approaches to generate test cases directly from the formalized requirements. But with these approaches the test coverage of the generated test cases are not high enough, especially for safety critical applications. Also inconsistencies within the requirements will not be detected. Therefore there are further approaches to synthesize a specification model based on a set of formal requirements. One advantage of these approaches are the identifications of inconsistencies during the synthesization of the specification model within the requirements. Also the achieved test coverage of the generated test cases is better and equivalent to the former described methods. The general process is described in Figure 2.11, an industrial use case is described in [80].

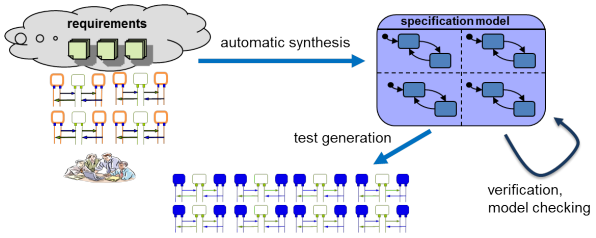

Figure 2.11: requirement based model synthesos for test generation purposes

Structural UML Model-based Techniques The Component diagrams depict how components are wired together to form larger components and/or software systems. They are used to illustrate the structure of arbitrarily complex systems. Component-based development (CBD) and object-oriented development go hand-in-hand, and it is generally recognized that object technology is the preferred foundation from which to build components Figure 2.12.

The class diagrams describe the structure of a system by showing the system's classes, their attributes, operations (or methods), and the relationships among objects. The class diagram is the main building block of object oriented modelling. It is used both for general conceptual modelling of the systematics of the application, and for detailed modelling translating the models into programming code. Class diagrams can also be used for data modelling. The classes in a class diagram represent both the main objects, interactions in the application and the classes to be programmed.

In the diagram, classes are represented with boxes which contain three parts. The top part contains the name of the class. The middle part contains the attributes of the class. The

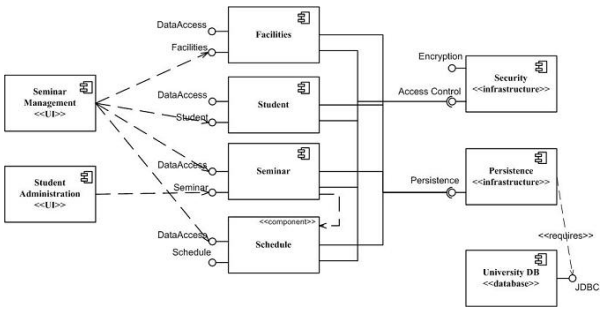

Figure 2.12: Component diagram

bottom part contains the methods the class can execute. Figure 2.13 depicts an example of a class diagram.

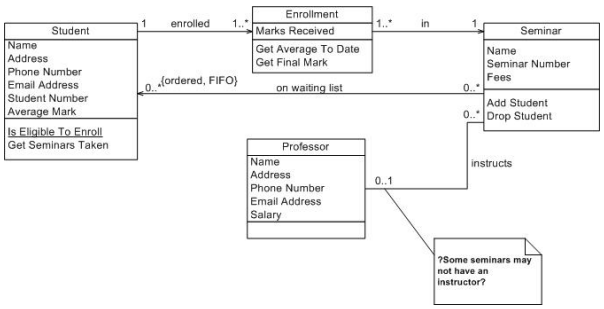

Figure 2.13: Class diagram

Class and object diagrams are used in "Automatically Generating Test Cases Using UML Structure Diagrams" [93] to generate test cases. The presented methodology accepts the application code as input and runs it to create a list called the class list which contains features of classes mentioned in the application; it then uses this class list to extract the features of each class as well as the relationships between them. Finally test cases are generated based on these features and relationships.

Class diagrams and state machines are used in "An Approach for Selective State Machine based Regression Testing" [53] to generate test cases that can identify the impact of changes made in class diagrams on the corresponding state machines and in turn on the test suite. The introduced methodology assumes the presence of test suite for the program under test. It presents a UML based selective regression testing strategy to identify changes and classify them. The changes are then classified as class-driven (obtained from class diagram) and state-driven (obtained from state machine). These changes are finally used to identify fault-revealing test cases.

The paper "Control flow analysis of UML 2.0 Sequence diagrams" [55] introduces the main seed of a class diagram-based methodology that generates test cases for regression testing. The former paper presents a control flow analysis methodology for sequence diagrams, which is based on defining formal mapping rules between metamodels. Then Object Constraint Language (OCL)-based mapping is made between sequence diagrams and Control flow graphs called Concurrent Control Flow Graphs (CCFGs), so as to ensure the completeness of the metamodels and allow their verification. This methodology is extended to fulfill the purpose of regression

testing where class diagrams are included to get more information. The current CCFG is renamed as Extended CCFG (ECCFG). The ECCFG is constructed using a sequence diagram and the corresponding class diagram. The extended methodology works by first having two versions of the same ECCFG, then comparing them to identify the changed nodes and arcs which are further used as input for test case selection and generation.

Class diagrams and Object Constraint Language (OCL) are used in "Automatic Test Cases Generation from Software Specifications" [20] to extract test cases from functional specifications by first transforming them to XML. Then extract specifications and use them to construct a class's hierarchy table which is used to create a classification tree. The tree is finally pruned to extract the final test cases.

Different Model-based Techniques An automatic model-driven technique is commonly used to generate test cases for Graphical User Interface-based applications (GUIs). The technique uses feedback from the execution of an initial test suite, which is generated using an existing structural event-interaction graph model of the GUI. During its execution, the run-time effect of each event on all other events determines event-semantic interaction (ESI) relationships, which are used to generate new test cases.

Mathematical models can be used as well to generate test cases even if other models will be involved. A technique that generates test cases from UML state charts using a mathematical basis is what "Formal Test Case Generation For UML State Charts" [57] proposed. The generation algorithm is written in a language which is a mix of process algebra and a simplified version of the lambda calculus. The final test cases might be represented again as state charts or sequence diagrams or just as code in a proper programming language. No assumption about their form is determined.

2.6.2 Test Cases Validation

In order to be able to claim that the generated test cases, based on previously defined software model, are better than others or even decide whether they are applicable or not, they must be first qualified for usage.

Quality of test cases depends on how well they cover the functionalities of the system under test and not only on their form. The test cases should be validated against known quality standards which determine their acceptable form as well as the degree of their functional coverage which in turn specifies their level of applicability. Many metrics have emerged and are being used to measure the quality of the test cases being generated like the time, cost, effort, complexity of generation, coverage criteria, and many others.

Coverage criteria are considered a set of metrics that are used to check the quality of test cases that are extracted from behavioural models. This metrics set contains many types of criteria and according to the UML model being used in generating the test cases, a certain criterion or many criteria are selected rather than the others. Some examples of the coverage criteria are:

- The branch coverage criterion; it is used with Control flow graphs.
- The full predicate and the condition coverage criteria; they are used to validate the test cases generated from state charts or communication diagrams.
- The all basic paths coverage criterion; it is used with activity diagrams-based techniques.

It can take several forms improving the quality of the test cases, such as decreasing the testing effort or time, decreasing the complexity or cost of the generation algorithms, increasing the functionality coverage as well as other quality and reliability issues. Also reducing the generated test cases can be a form of optimization.

Test case reduction and optimization techniques

Reduction of the number of test cases is a major target of some approaches such as the work presented in "Evolving the Quality of a Model Based Test Suite" [54] and "Optimizing for the Number of Tests Generated in Search Based Test Data Generation with an Application to the Oracle Cost Problem" [61]. The former approach is an evolutionary-based algorithm that presents a novel model-based test suite optimization technique involving UML activity diagrams by explicating the test suite optimization problem as an Equality Knapsack Problem. The latter technique uses an algorithm depending on various testing techniques which are: Evolutionary testing, Genetic algorithms, and the Search based testing. It covers the branch coverage of functions as a unit testing model. Another technique presented in "A Black-Box Test Case Generation Method" [76] proposes a requirement prioritization process during a test case generation process by introducing a method that generates multiple test suites while minimizing the number of test cases in them using UML scenarios.

A model-based regression testing approach that uses Extended Finite State Machine (EFSM) is presented in "Model Based Regression Test Reduction Using Dependence Analysis" [75]. It is used to reduce the regression test suites. The modified parts of the model are tested using selective test generation techniques, but still the size of regression test suites may be very large. As a result, the approach automatically identifies the differences between the original model and the modified model as a set of elementary model modifications. For each elementary modification, regression test reduction strategies are used to reduce the regression test suite based on EFSM dependence analysis.

Dynamic symbolic execution is a structural testing technique that systematically investigates feasible paths of the program under test by running the program with different test inputs. Its main goal is to find a set of test inputs that lead to the coverage of particular test targets. Many techniques include Dynamic Symbolic Execution (DSE) technique in test case generation. However, these DSE techniques, as claimed by Reggae: "Automated Test Generation for Programs using Complex Regular Expressions" [79], cannot generate high-covering test inputs for programs that use complex regular expressions due to the huge search space. To handle this problem, an approach is proposed named Reggae that reduces the search space of DSE in test generation, thus generating test data with higher branch coverage. However practically the number of feasible paths explored may explode, thus another search strategy called Fitnex, was proposed in "Fitness-Guided Path Exploration in Dynamic Symbolic Execution" [112], that uses state-dependent fitness values which are computed using a fitness function to guide the path exploration.

A technique is introduced in Reassert: "Suggesting repairs for broken unit tests" [43] where a tool called ReAssert is built to repair test cases that have failed due to changes that have been made in the requirements which cause changes in the code. It makes changes to the test case's code to enable the passing of failed tests. It also displays the repaired and failing test code for the user to confirm the changes or make further modifications on them. However ReAssert has some limitations, like its ability to only repair about 45% of failures in open-source applications. Also ReAssert suggests a suboptimal repair, which means that a more

useful repair can be possible. Moreover, if a failing test modifies expected values, creates complex expected objects, or has multiple control-flow paths, then ReAssert cannot determine what expected values need to be changed and in what way. Then a modification comes on the ReAssert in "On test repair using symbolic execution" [42] to introduce a symbolic test repair which repairs more test failures and provides better repairs. It is a technique that uses the symbolic execution to change the literals in the test code. This technique can overcome some of ReAssert's limitations mentioned previously. It is also developed in java. Pex is another tool which can be used for the same aim but it is developed for .Net applications, cf. "Pex - White Box Test Generation for .NET" [103].

2.7 Simulation

According to Banks et al. [26], simulation imitates operations of real-world processes or even entire systems over time. For this purpose, similar to formal methods using mathematical analysis – e.g., presented in previous subsections – simulation requires a model of the considered system. This model should represent the system as accurate as possible related to the focused properties which are aimed to be verified respectively validated. In practice, simulation is known as an appropriate approach to compare alternative designs as well as to optimize a particular design [60]. Furthermore, simulation supports estimation of best and worst case execution times (BCET and WCET) that are required as input for verification and validation steps. Last but not least, simulation enables verification and validation of system behavior for particular situations by considering dedicated input parameters.

Figure 2.14 depicts the comparison of lower/upper execution time bounds obtained by analytical methods, minimal/maximal observed executions times from simulation or hardware measurements as well as the actual BCET/WCET. While analytical methods usually provide very pessimistic estimations resulting, e.g., in overestimated WCETs, simulation techniques enable more realistic approximation of WCETs, i.e. estimations resulting from measurements like simulation tend to be the normal case. This is because simulation results depend on the considered input data that have to be well chosen to cover most common system behavior. But this processing also implies that one cannot guarantee to cover the worst case. Consequently, approximations resulting from simulation are usually underestimated with reference to the actual WCET.

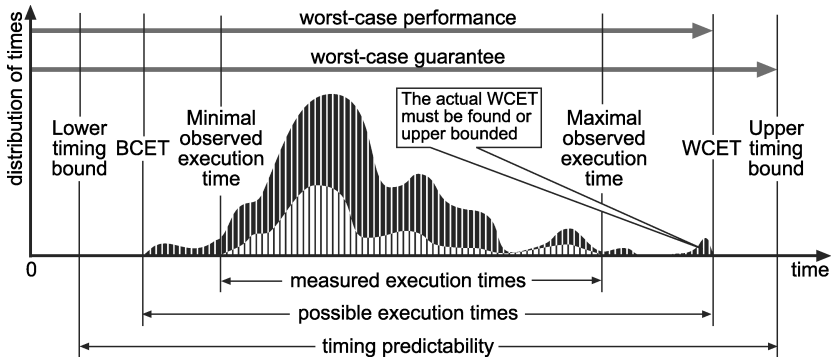

Figure 2.14: Simulation accuracy: Comparison of lower/upper execution time bounds obtained by analytical methods, minimal/maximal observed executions times from simulation or hardware measurements as well as the actual BCET/WCET. (cf. [111]).

2.7.1 Simulation Techniques Overview

Various simulation techniques are known that enable system simulation on different levels of abstraction. This enables verification of system designs during early design phases and thus provides the opportunity to support, e.g., design space exploration at system- and micro-architecture levels [59]. Figure 2.15 shows the comparison of system level analytical models as well as the simulation techniques abstract performance simulation, instruction set simulation, cycle accurate simulation and HDL/RTL simulation with reference to their accuracy and required evaluation/implementation time.

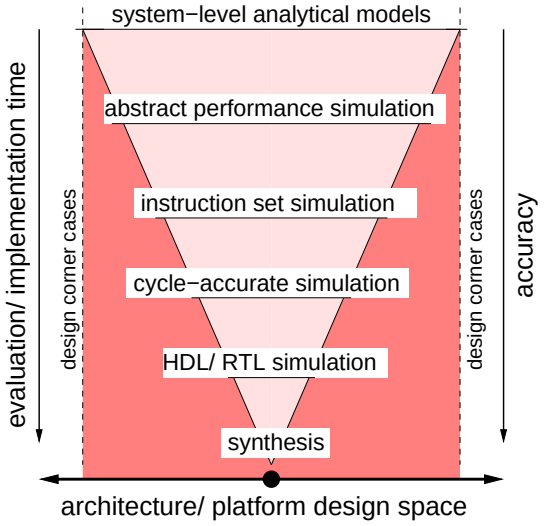

Figure 2.15: Simulation techniques: Comparison of different simulation techniques regarding accuracy and evaluation/implementation time (cf. [59]).

In the following, we give a more detailed description of discrete-event simulation – as an example for abstract performance simulation – and instruction set simulation that covers instruction as well as cycle-accurate simulation.

2.7.2 Discrete-Event Simulation

Discrete-event simulation is an efficient way to simulate systems whose states can only change at discrete points in time. Jumping in time from one event occurrence to the successive event occurrence, discrete-event simulators utilize the fact that inbetween the occurrence of two consecutive events a system cannot change its states [60]. This way, simulators consider only those points in time when a system might change its states and thus saves time spent for simulation. Figure 2.16 shows the principle of discrete-event simulation.

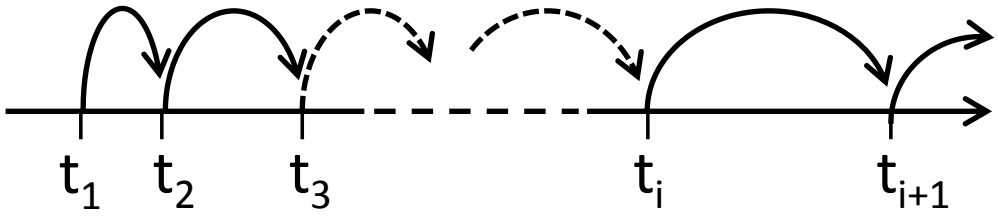

Figure 2.16: Discrete-event simulation: System states can only change at discrete points in time and thus enables simulators to step from one event occurence to the successive event (cf. [60]).

Each time an event occurs, new so called event notices can be generated. An event notice represents future events and consists at least out of

Time defining the point in time when this event will occur and

Type specifying the kind of this event [60].

Discrete-event simulators usually hold a future event list to manage these event notices. This future event list is ordered by the points in time when events occur next. An overview of further components usually contained by a discrete-event simulator is given by Tab. 2.4.

Term	Description
System state	Set of variables
Clock	Provides current (simulated) time
Future event list	Used to manage future events
Statistical counters	Set of variables to store information about system performance
Initialization routine	Initializes the simulation model and simulation clock
Timing routine	Retrieves the next event from future event list and sets the clock to the occurrence time of the event
Event routine (also called handler)	Called when a particular event occurs during simulation; typically, an event routine is defined for each event type

Table 2.4: Components of a discrete-event simulator (cf. [60])

In general, a discrete-event simulator processes as follows: It starts with an initialization routine that initializes all entities, state variables, and the clock of the simulator providing the current time during a simulation run. Then, events are processed in a loop: The simulator gets

the next event from the future event list and sets the clock to the occurrence time of this event. Depending on the event's type, the corresponding event routine is called which can

- change the state variables and entities,
- update some statistical counters, and
- generate new event notices that are added to the future event list.

This loop is processed until the termination condition becomes true. Finally, the simulator creates some output as, e.g., a trace or statistics describing simulated system characteristics. Figure 2.17 shows the usual work flow of a discrete-event simulator by means of a state machine.

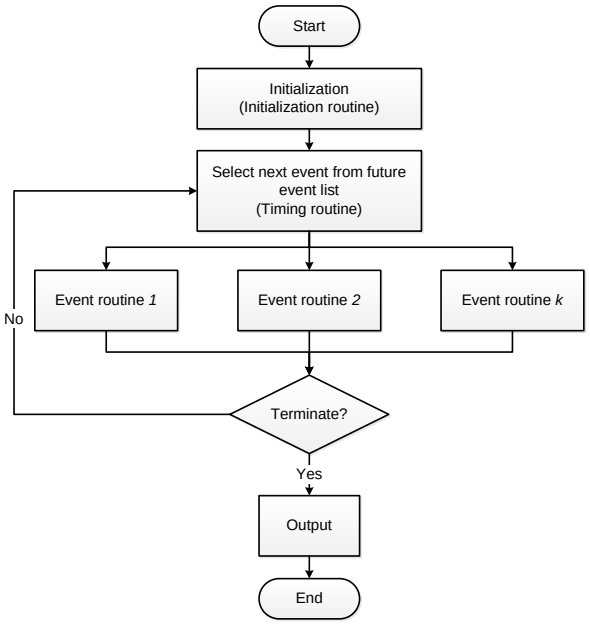

Figure 2.17: State machine of a discrete-event simulator (cf. [60]).

2.7.3 Instruction Set Simulation

As the name implies, instruction set simulation is based on the instruction set of a target hardware architecture. Here, target hardware refers to the hardware platform where the considered software system is supposed to be deployed onto. For this purpose, the instruction set of the target hardware is simulated by a simulator software executed on a host system. A host system is usually a computational powerful system – e.g., a desktop PC – whose instruction set architecture differs from that of the target hardware. Therefore, an instruction set simulator provides a mapping of target hardware instructions to a single instruction or sequence of instructions of the host system. This way, instruction set simulators enable simulation of target code.

However, it is important to note that instruction set simulation is based on some kind of hardware models and thus usually imply some abstraction resulting in a loss of accuracy between

simulated and real target hardware. Consequently, instruction set simulation does not guarantee that system behavior of the simulated hardware is the same as of real target hardware. Alizai et al. [21] distinguish two kinds of instruction set simulations:

- *Cycle accurate* instruction set simulation
- *Instruction accurate* instruction set simulation.

As implied by the names, cycle accurate instruction set simulators operate on granularity of cycles of a processor core and instruction accurate ones operate on granularity of instructions. Nevertheless, both kinds of instruction set simulators in general operate as follows [21]: An executable resulting from software development is loaded by an instruction set simulator into its memory. Afterwards, the simulator processes a loop by first fetching an instruction of the executable from memory. Next, this instruction is decoded for the targeted instruction set architecture and finally executed on the host machine to simulate the behavior of the target processor. Figure 2.18 depicts this general operation of instruction set simulators.

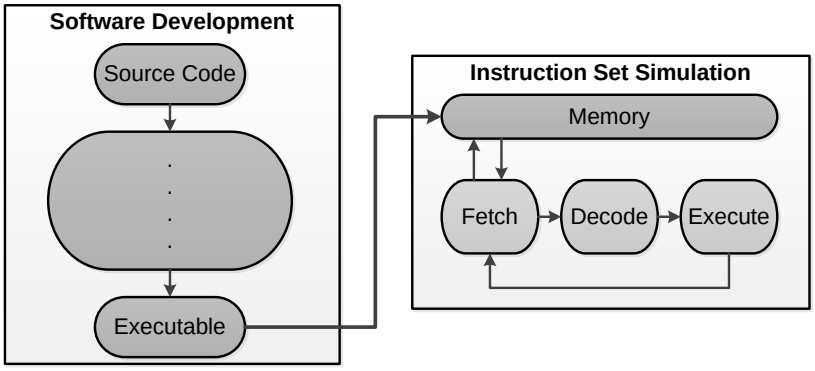

Figure 2.18: General principle of an instruction set simulator (cf. [21]).

As instructions differ with reference to their execution times, instruction accurate simulation allows simulation of functional system behavior, but timing behavior is neglected in general. However, there are approaches to include timing to instruction set simulations. For instance, instructions could be annotated by an execution time to respect their different timing requirements. Cycle accurate simulations provide the capability of timing analysis as execution times can be computed based on the number of cycles counted during simulation. But as already stated above, cycle accurate instruction set simulation requires a much more detailed hardware model to reach this accuracy.

2.8 Product Line Analysis

The automotive industry and others use product lines [39, 106] for developing complex products combining system parts of different engineering disciplines like mechanical engineering, electrical engineering and software engineering [97]. Product lines [39] are a set of products or systems sharing a common set of features. Through the selection of variants, different products can be built out of the product line. Building such product lines (also called product family) requires three essential activities (see Figure 2.19).

Figure 2.19: Essential Product Line Activities [39]

First, the platform also called core assets has to be developed, which contains all common parts of the product line. Therefore, different inputs need to be considered in order to develop the core assets as well as a production plan (Figure 2.20).

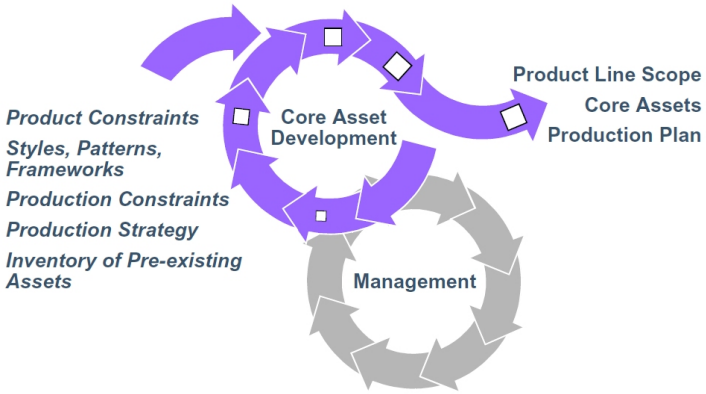

Figure 2.20: Core Asset Development [39]

The second activity is the product development, which use the output of the first activity for building individual products. Furthermore, the requirements for the individual product are an input, which are used to identify necessary adaptations (variants) of the platform. Both activities need to be highly managed through the third. When setting up a product line three approaches are possible [77]:

- *Proactive*: In the proactive approach, all products and variants are planned and developed in advance, which is like the waterfall approach to conventional software.
- *Reactive*: The development of the product line starts with a few variants and is extended incrementally (more or less as in a extreme programming approach).
- *Extractive*: In the extractive approach, one or more existing products are used to form the bases for the new product line.

While developing such product lines *feature models* [72] are used for describing these common and variable parts of the system. A feature model consists of a hierarchically arranged set of features connected through different types of associations. During the product configuration (e.g. described in [22, 41, 36]), the model can be used for creating valid combinations of features. For this purpose, features are connected to software development artifacts e.g. software components.

In AMALTHEA4public, we focus on the development of product lines including all three approaches. To enable the development of variable systems, it is necessary to integrate them into the development process. To add the aforementioned support to the existing development processes, the process has to be extended by introducing further steps regarding variant development. Because of the nature of a product line, the whole process consists of two fundamental steps, the development process and the configuration process. While the development process includes steps to identify requirements and to develop the product line including the required features, the configuration process includes the product generation based on a customer's needs.

The problem is that product lines have a long lifetime, while system parts of it may have different life-cycles so that parts needs to be changed or replaced at certain stages. This applies in particular to the software evolution [58]. If a product line feature or a configuration changes - same as if a requirement changes - the software components (and in some cases the software architecture) has to be changed to reflect the changed functionality. Some changes even result into a changed target hardware platform. Such changes happen frequently during a software development project. Especially late changes demanded by the customer are a problem.

Because of these changes during the long lifetimes that exist particularly in the automotive industry, analysis methods are important in order to identify errors in the models. For this purpose various automatic methods have been developed in the past [32], e.g. for the identification of dead features. By a systematic identification of all locations that are affected by a change (*Change Impact Analysis*), it would be possible to minimize the errors caused by changes. This approach focus on uncovering errors, instead of avoiding them. Nevertheless, also effective methods for change impact analysis of product lines are necessary.

2.8.1 Feature Models

A central element for the representation of software variants (common and variable parts) are feature models [72]. Each feature model consists of many features, which represent stakeholder requirements or product characteristics [38, 27, 41]. These features are arranged in a tree structure and may have dependencies to other features. Supplementary each feature has a semantic which indicates the meaning within a group (optional, mandatory, or, alternative). The feature itself is associated with a concrete software component or software artifact, so that it is possible to produce software based on a selection of features and their dependencies. A defined and valid selection of features represents one configuration (software product) of the software product line.

After the introduction of feature models in 1990 as part of the FODA [72], many extensions have been proposed in the following years. One remarkable extension are the so called Cardinality-Based Feature Models. Features and feature groups can receive a cardinality which defines a limitation that will restrict the selection during the configuration process. The Mandatory and Optional elements can be understood as special cardinalities. Therefore the cardinality [1..1] would define a mandatory and [0..1] an optional element [40]. In combination with feature groups, Alternative and Or groups can be expressed as cardinalities of [1..1] and [1..*]. Other

extensions are for example, modularization, attributes, relationships, feature categories, and annotations.

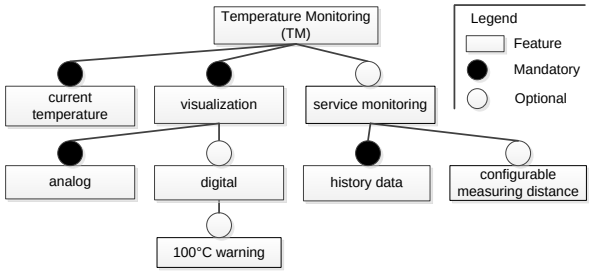

Figure 2.21: Software variants of the temperature monitoring (TM) example

Figure 2.21 shows an example feature model of a temperature monitoring (TM) system. In any case, the system measures the actual temperature and visualizes it in an analog way. Optionally, it is possible to choose a digital visualization as well as a warning function, which displays a warning, if the temperature rises above 100 degrees Celsius. Another optional variant is the service monitoring capability. This variant allows to establish a web connection to the system via an Ethernet interface. Using this interface, the latest temperature values can be displayed as well as the current sample rate. Another variant can extend this interface, which allows the user to change the sample rate.

2.8.2 Feature Model Analysis

The analysis of feature models uses information of the model in order to automatically perform analysis operations and thus check the models. Over time, a variety of operations was developed [32, 109, 33].

Analysis operations

- Void feature model
  It is checked whether it is possible to create a valid product from a feature model. The feature model is void when no product can be produced.
- Valid product
  This operation checks whether a set of features is a valid set of a Feature model.
- All products/Number of products
  This operation determines all the possible products of a feature model. It is also possible to determine only the number of products.

Detection of anomalies

- Dead features
  A feature that cannot be part of any product.
- False optional features
  A feature that is optional, but has to be part of any product.

- Wrong cardinalities
  A Cardinality, which cannot be instantiated (mostly due to cross-dependencies).

Optimization methods

- Core features
  Determines all features that have are part of all valid products.

2.8.3 Change Impact Analysis

A Change Impact Analysis is defined by Bohner and Arnold as "identifying the potential consequences of a change, or estimating what needs to be modified to accomplish a change" [34]. Furthermore, a CIA will allow determining variants, which will not be supported by the new system parts. Based on the operations in [35], the following changes can occur in a product line.

- **Modify**
  Modifying affects the feature model as well as the component model. On the one hand, the tree structure of the feature model can be changed by modifying a feature type, shifting or merging of features. On the other hand, the feature and its realizing components can be modified. Furthermore, changes in the target or source of cross-tree constraints are possible.

- **Delete**
  During the deletion of elements, they will be completely removed from the system family, so that related elements need to be revised. This applies for other features, constraints and software components, too.

- **New**
  Similar to deleting elements, it is necessary to determine related features and components during the analysis while adding of new constraints and components. Adding new features is a special case. It is not possible to identify related elements, so that a prediction is required.

In recent years, various approaches have been published [86, 85, 46, 19], based mainly on traceability links or analyzes produced products before and after a change.

3 State of the art V&V tools and frameworks

This section aims to describe some existing model verification tools and frameworks. This tool list is aligned with verification and validation methods seen above.

3.1 Informal verification tools and frameworks

Informal reviews are applied many times during the early stages of the life cycle of software development. The most important thing to keep in mind about the informal verification is that usually they are not documented and there are not specific tools for informal verification, so it is performed using non specific ones. Some examples of informal methods are detailed next.

3.1.1 Face Validation

Analyzing the accuracy of a chess bot simulator's response to user input to verify that the A.I. is reacting in a logical manner. Or the accuracy of a train simulator's response to control inputs can be evaluated by having an experienced train operator driving the simulator through a range of maneuvers.

3.1.2 Walkthrough

A software development team reviewing a product before the final product is sent for approval by the customer.

3.1.3 Desk Checking

Any programmer who develops software participates in the informal method of verification known as desk checking. Debugging software as it is being developed is a form of desk checking. The developer sets breakpoints or checks the output from the model to verify that it matches the algorithms developed in the conceptual model.

3.1.4 Turing Test

Cleverbot [1] is an application that interacts with people by responding to questions and learning from replies. Testing of Cleverbot is best completed by using a Turing Test. Interacting with Cleverbot allows the user to analyze whether or not they can distinguish between the fact that it is actually just code responding to them, or if they believe that it is another human.

3.2 Static verification tools and frameworks

3.2.1 ModelJUnit

ModelJUnit [8] is an open source Java library that extends JUnit [5] to support model-based testing. The models are written in Java. ModelJUnit allows the user to write simple finite state machine (FSM) models or extended finite state machine (EFSM) models as Java classes, then generate tests from those models and measure various model coverage metrics.

3.2.2 MBT-Tool

MBT-Tool [7] specifically aims to enhance the test design process for communication protocols, e.g. used in smart card based systems. Replacing manual steps with automated methods based on UML models and suitable algorithms enables more comprehensive test coverage and extended test depth. MBT-Tool covers the following test methods:

- Modeling of static test methods: Testing of data structures as expected on the card (i.e. machine-readable passport or national ID card) and manipulated data as it may be sent to a reader or terminal.
- Modeling of dynamic test methods for protocol testing (i.e. PACE), simulation of card behavior and simulation of terminal behavior.
- Meta modeling of protocols and static data in UML.

3.3 Dynamic verification tools and frameworks

3.3.1 MaTeLo

MaTeLo [6] implements a Model-Based Testing approach in a user-friendly environment. Starting from application usages, business requirements or user stories, testers design models able to automatically generate optimized test suites. These test suites can be exported either to automatic execution tools or to test management tools for manual execution.

Business requirements or User Stories are created into MaTeLo or imported from a requirements manager tool (HP ALM QC, Doors, Test Link,...).For each test step, stimulation data is generated from equivalence classes. Output test data set (Test Oracle) can be computed from treatment functions or from data sources (files, DB, Web Services, ERP...), or from external tool calculators (Matlab, Scilab) or by using Python functions.

3.3.2 JUMBL

JUMBL [91] is a Java class library and set of command-line tools for working with usage models. The JUMBL supports construction and analysis of models, generation of test cases, automated-execution of tests, and analysis of testing results. The JUMBL provides support for every phase of statistical testing. Models can be constructed from libraries of common components using a language such as TML and are then analyzed to determine their stochastic properties. The JUMBL can generate tests in various ways and convert test cases into automated test information. The results of testing can be analyzed to determine the expected system performance.

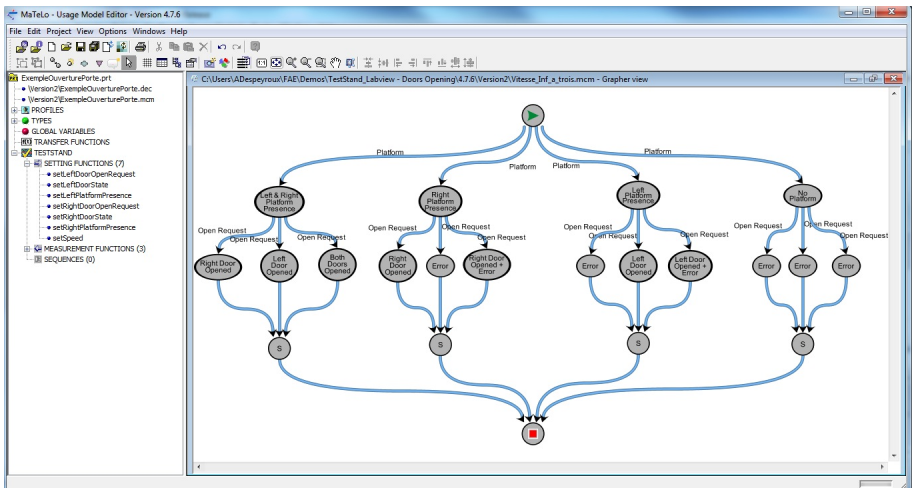

Figure 3.1: MaTeLo model editor

3.3.3 Spec Explorer

Spec Explorer [10] is a tool that extends Visual Studio for modeling software behavior, analyzing that behavior by graphical visualization, model checking, and generating standalone test code from models. The behavior is modeled in two ways: by writing rules in C# (with dynamic data-defined state spaces) and by defining model scenarios as action patterns in a regular-expression style. One of Spec Explorer's major features is the ability to compose models written in these two styles. This technique enables users to slice out test cases from large state machines to achieve test purposes by defining relevant scenarios, thus tackling the notorious state-space explosion problem that is so pervasive in model-based testing. Spec Explorer also supports combinatorial interaction testing with a rich set of features.

3.3.4 TorX

The TorX tool [14] is a prototype testing tool for conformance testing of reactive software. The tool requires a real implementation and a formal specification of that implementation. The specification describes the system behavior that the real implementation is allowed to perform. The TorX tool checks the correct behavior of a real implementation during its execution based on the formal specification.

3.3.5 Uppaal TRON

Uppaal TRON [15] is a testing tool based on the Uppaal engine. It is suited for black-box conformance testing of timed systems, mainly targeted for embedded software commonly found in various controllers. Tests are derived, executed and checked simultaneously while maintaining the connection to the system in real-time.

3.4 Formal verification tools and frameworks

Formal verification is integrated in different areas through frameworks that enable reasoning about the correctness of the developed software application. Throughout the last two decades a certain variety of frameworks and tools for Model Checking for both general as well as specific domains were introduced.

Currently there are many tools of model checking. Most of them require that the model fits their own specification language. Furthermore, many checking tools are directed to a single temporal logic. In the context of Model Checking usually temporal logic is used to specify the property to verify (e.g. CTL, LTL, CTL *, ...).

3.4.1 SPIN

Spin [11] is an open-source software verification tool that can be used for the formal verification of multi-threaded software applications. The tool supports a high level language called Promela to specify systems descriptions.

Verification models are focused on providing the correctness of process interactions, and they attempt to abstract as much as possible from internal sequential computations. The processes refer to system components that communicate with each other in terms of asynchronous message passing through buffered channels (including access to shared variables or with any combination of these).

Furthermore, it provides direct support for use of embedded C code as part of model specifications. This makes it possible to directly verify implementation level software specifications and as a logic engine to verify high level temporal properties.

The basic structure of Spin is to start with the high level specification of a concurrent model or distributed algorithm, using Spin's graphical front-end XSpin. After fixing syntax errors, interactive simulation is performed until confidence is gained.

Otherwise, Spin is also used to generate an optimized on-the-fly verification program from the high level specification. This verification is compiled, with possible compile-time choices for the types of reduction algorithms to be used, and executed. If the design does not behave as intended, these can be fed back into the interactive simulator and inspected in detail to identify and remove their cause.

3.4.2 UPPAAL

Uppaal [30, 44] is an integrated tool environment for modeling, simulation and verification of real-time systems, developed jointly by Basic Research in Computer Science at Aalborg University in Denmark and the Department of Information Technology at Uppsala University in Sweden. It is appropriate for systems that can be modeled as a collection of non-deterministic processes with finite control structure and real-valued clocks, communicating through channels or shared variables. Typical application areas include real-time controllers and communication protocols in particular, those where timing aspects are critical.

Uppaal consists of three main parts: a description language, a simulator and a model-checker. The description language is a non-deterministic guarded command language with data types (e.g. bounded integers, arrays, etc.). It serves as a modeling or design language to describe system behavior as networks of timed automata extended with clocks and data variables. The simulator is a validation tool which enables examination of possible dynamic executions of a

system during early design (or modeling) stages and thus provides an inexpensive mean of fault detection prior to verification by the model-checker which covers the exhaustive dynamic behavior of the system. The model-checker can check invariant and reachability properties by exploring the state-space of a system, i.e. reachability analysis in terms of symbolic states represented by constraints.

To facilitate modeling and debugging, the UPPAAL model-checker may automatically generate a diagnostic trace that explains why a property is not satisfied by a system description. The diagnostic traces generated by the model-checker can be loaded into the simulator for visualization and investigation.

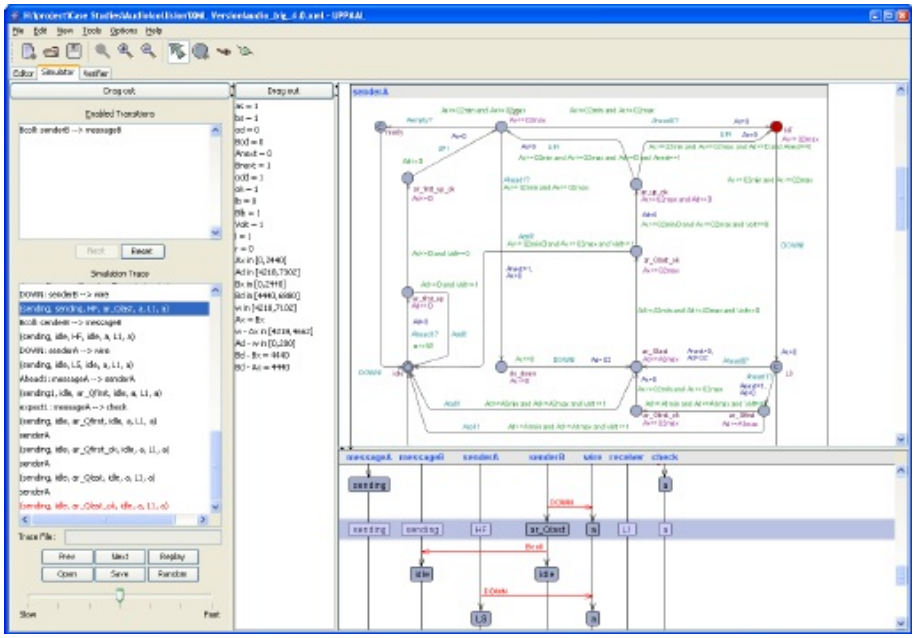

Figure 3.2: The simulator in the UPPAAL GUI

3.4.3 NuSMV

NuSMV [9] is a reimplementation and extension of SMV symbolic model checker, the first model checking tool based on Binary Decision Diagrams (BDDs).The tool has been designed as an open architecture for model checking. It is aimed at reliable verification of industrially sized designs, for use as a backend for other verification tools and as a research tool for formal verification techniques.

NuSMV supports the analysis of specifications expressed in CTL and LTL. User interaction is performed with a textual interface, as well as in batch mode.

NuSMV 2, version 2 of NuSMV, inherits all the functionalities of NuSMV. Furthermore, it combines BDD-based model checking with SAT-based (Boolean Satisfiability Problem) model checking.

3.4.4 Java PathFinder

Java PathFinder (JPF) [4] is an open source explicit-state model checker for Java programs distributed under GPL license. It explores all executions that a given program can have due to different thread interleavings and nondeterministic choices. JPF implements a backtracking Java Virtual Machine (JVM) that executes bytecodes using a special representation of JVM states. This special representation enables JPF to quickly store, restore, and compare states; it is crucial for making the overall state exploration efficient. However, the special representation creates additional overhead during each execution.

The core of JPF is a virtual machine (VM) which runs on top of the Java VM in order to enable model checking of user programs. JFP is mainly used for:

- Exploring alternative execution
  - Scheduling sequences – concurrent applications.
  - Variations in input data.
  - Environment events.
  - Program control flow choices.
- Execution inspection

The core of JPF (jpf-core project) discovers defects without the need for specifications of any program properties. This includes: deadlocks, race conditions, uncaught exceptions. Manifestation of nonfunctional properties should not occur in any application.

The exploration of the set of inputs and an optional bound on the length of program execution allows to determinate all executions (up to the given bound) which the program can have due to different thread interleavings and nondeterministic choices. JPF can generate those executions that violate a given (temporal) property in terms of example traces.

3.4.5 Symbolic PathFinder

Symbolic PathFinder (SPF) [12] combines symbolic execution with model checking and constraint solving for test case generation in Java bytecode programs. Programs are executed on symbolic inputs representing multiple concrete inputs. Values of variables are represented as numeric constraints, generated from analysis of code structure. These constraints are then solved to generate test inputs guaranteed to reach that part of code.

SPF is part of the Java PathFinder verification tool-set Figure 3.3, a freely available open-source project. It uses the instruction factory class SymbolicInstructionFactory to build bytecode instructions that manipulate symbolic values and expressions. SPF stores these symbolic values in special "attributes" associated with the program data (variables, fields and stack operands).

The symbolic execution of conditional instructions leads to the exploration of distinct program paths. SPF relies on the JPF-core framework to systematically explore the different choices of symbolic execution paths as well as thread interleavings. The choices are explored exhaustively using a mechanism of Java PathFinder-core known as "choice generators" (e.g. PCChoiceGenerator) for the construction of path conditions. A path condition is associated with each choice generated and is checked using a decision procedure or a constraint solver. If the path condition becomes unsatisfied, JPF is automatically instructed to backtrack.

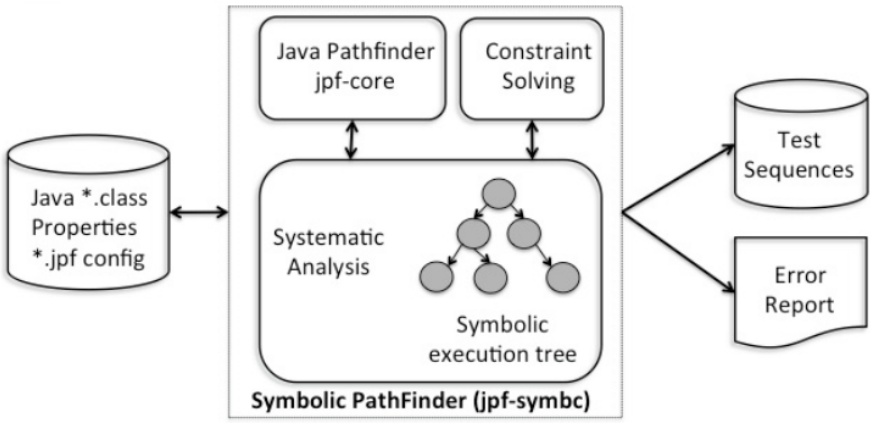

Figure 3.3: Symbolic PathFinder

The Symbolic listeners gather and display information about the path conditions generated during the symbolic execution. To limit the possibly infinite (symbolic) search state space that results from analyzing programs with loops, the user needs to provide a limit on the model checker's search depth or on the number of constraints encoded in the path condition.

3.4.6 MechatronicUML

The section introduces the MechatronicUML method and especially it's compositional verification approach. Major parts of this summary are a shortened version based on [62, 28].

MechatronicUML[1] provides a method and tools for the model-driven development of safe software for interconnected mechatronic systems [28, 90]. A strong focus is on the coordination and communication between systems and sub-systems as well as on the early verification of the modeled software. MechatronicUML defines a formal modeling language which uses concepts of the UML [84] and follows a component-based approach (cf. [102]). This enables a scalable compositional verification of the system model, utilizing Uppaal as the underlying model-checker[62]. The complete method specification can be found in [28].

Here, we use the RailCab system to introduce the basic concepts of MechatronicUML and present its compositional verification approach [63]: The vision of the RailCab project is a railway transportation system where autonomous vehicles, called RailCabs, travel on a track system. The track system is subdivided into sections including switches and railroad crossings. In this system, collision avoidance on track has to be realized by the system itself. In particular, collision avoidance requires communication of the RailCab with various track systems or other RailCabs because sensors cannot detect other RailCabs if they are hidden behind a bend or some other obstacle. In our example, each RailCab must query an upcoming track section whether it is allowed to enter this track section. Considering the situation shown in Figure 3.4, both RailCabs want to enter the switch `ts3`. They communicate with the switch by adhering to a communication protocol. This protocol is safety-critical because it must ensure that only one of the two RailCabs may enter the `ts3` at a time to avoid a collision. Therefore, the communication protocol must obey real-time constraints to ensure that the RailCab comes to a stop before the switch if it is not allowed to enter. Hence, we need to apply formal verification

[1] http://www.mechatronicuml.org/en/index.html

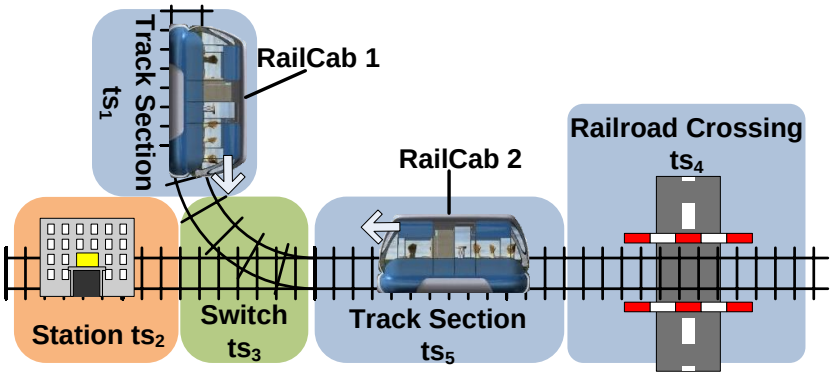

Figure 3.4: Example: Two RailCabs Approaching a Switch and a Railroad Crossing Afterwards

to guarantee safety of the RailCab system. However, the state space of the software models of the RailCab and the track system is too large to be assessed by standard model checking approaches based on timed automata [62]. The approach to tackle this challenge is introduced in Figure 3.4.6.

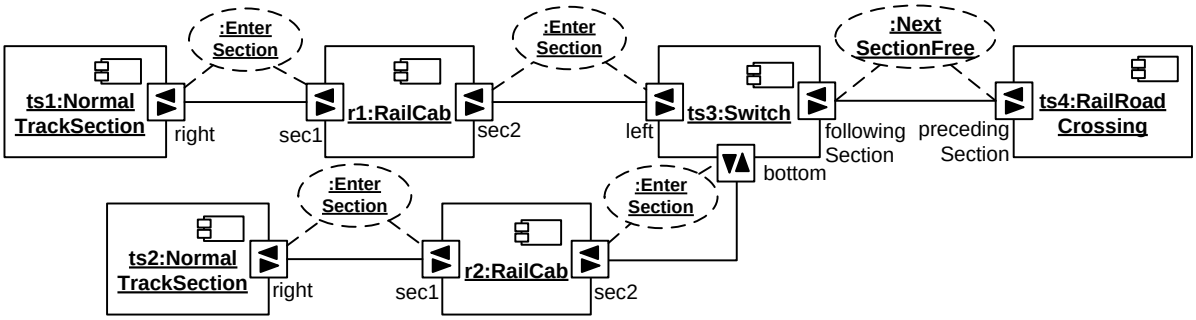

Figure 3.5: Software Architecture (Instance Configuration) of the Railcab Scenario, modeled in MechatronicUML

Figure 3.4 shows a part of the software architecture for the presented Railcab Scenario, modeled in MechatronicUML. Each entity of the scenario is modeled in terms of a software component (e.g. `ts1:NormalTrackSection`, `r1:RailCab`...; cf. 3.4.6). These are represented by rectangles with component symbols similar to UML. Connections and interactions between software components are modeled in terms of Real-Time Coordination Protocols (cf. 3.4.6). These are represented by ovals (e.g. `:EnterSection`) and attached to the components via ports (smaller rectangles with arrows). Both the component and protocol behavior are modeled in terms of Real-Time Statecharts (cf. 3.4.6). Please refer to the MechatronicUML method specification for a more detailed description of all language elements [28].

MechatronicUML development process

This section gives a brief overview of the MechatronicUML development process (cf. Fig. 3.6). Please refer to [28] for a more detailed description. Here, we focus on step 2 which is the

design of an *platform-independent software model* (PIM): Based on a set of formal requirements [2], the components of the overall system structure are specified in step 2.1. Each component is a software entity that encapsulates a part of the system behavior (cf. section 3.4.6). In step 2.2, the interactions between the components are specified in terms of Real-Time Coordination Protocols (cf. section 3.4.6). Real-Time Coordination Protocols specify a contract between components, in particular allowing to specify the message exchange as well as real-time properties. Also, the first integrated analysis step is performed: The safety and liveness properties of each Real-Time Coordination Protocol are verified formally using timed model checking (cf. section 3.4.6). In step 2.3, based on the Real-Time Coordination Protocols of step 2.2, the internal component behavior is derived (i.e. refined from the protocol behavior), again in terms of Real-Time Statecharts. The correct refinement is formally verified by a refinement check (cf. section 3.4.6). Afterwards, the component is then formally verified for deadlock freedom to ensure that all ports of a component communicate safely with each other. These three verification steps make up to overall compositional verification approach of MechatronicUML. In Step 2.4, the model of the discrete behavior is integrated with the feedback controllers of the mechatronic system. The formal verification of a correct integration of the controllers, however, is not feasible for real-world systems (cf. [28]).

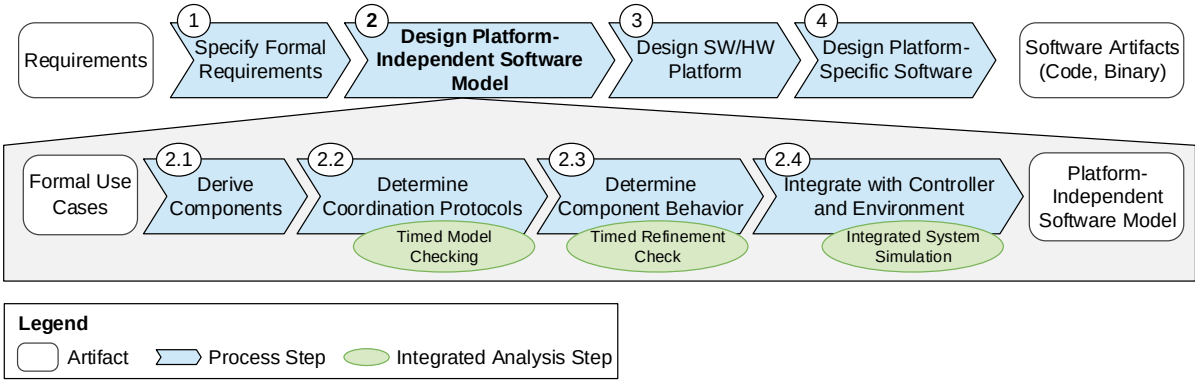

Figure 3.6: MechatronicUML Development Process (from: [28])

Component Model

Each component in MechatronicUML encapsulates a part of the software and defines a set of external interaction points, which are called ports. As an example, Fig. 3.7 shows the components of our Rail-Cab example. Each system element (`RailCab`, `normal track section`, `switch`, and `railroad crossing`) is represented by one component. Component `RailCab` has two ports: `sec1` communicates with the section the RailCab is currently on; `sec2` communicates with the section the RailCab will drive on next. The components `NormalTrackSection`, `Switch` and `RailRoadCrossing` have a port `left` to communicate with RailCabs that drive from left to right and a port `right` to communicate with RailCabs that drive the opposite way. Component `Switch` must also be able to communicate with a third Rail-Cab. Therefore, the component has a third port `bottom`. Additionally, the components `Switch` and `RailRoadCrossing` shall communicate with each other. Thus, `Switch` has a port `followingSection` and `RailRoadCrossing`

[2] these are modeled in terms of Modal Sequence Diagrams

has a port `precedingSection`. The component model is structured hierarchically: Components are either atomic components, i.e., they are implemented directly (atomic components) or they are structured components, i.e., they are decomposed into further components. Only atomic components have a behavior. For a detailed description of the component model, please refer to [28]. Feedback controllers are integrated as continuous components into the component model. MechatronicUML provides no behavior specification for feedback controllers. Instead, it is assumed that this behavior is specified by a control engineering tool like CamelView or MATLAB/Simulink.

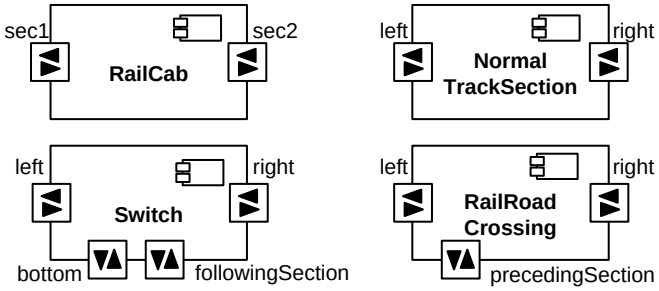

Figure 3.7: Components of the RailCab Scenario (from: [62])

Real-Time Coordination Protocols

Real-Time Coordination Protocols are abstract protocols on application level that define the coordination between a pair of communicating roles and a connector that connects these roles. The communication is always asynchronous and message-based, i.e., the sender can send its message independent of the receivers state, the sender of a message does not block while sending, and the receiver does not block while receiving.

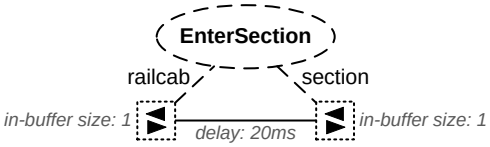

Figure 3.8: Coordination Protocol "`EnterSection`" (from: [62])

Figure 3.8 shows an example of a Real-Time Coordination Protocol named `EnterSection`. It consists of the roles `railcab`, which represents a RailCab, and `section`, which represents various kinds of track sections, e.g., the sections of our running example: a normal track section, a switch, and a railroad crossing. Both roles may send and receive messages to realize this coordination (indicated by the two black triangles). The purpose of this protocol is to enable a coordination for traversing a section in a safe and efficient manner[62]. The behavior of the associated roles is specified in terms of Real-Time Statecharts (cf. 3.4.6). Each role which receive messages has a message buffer for incoming messages, which we call in-buffer. The in-buffer always enqueues messages in FIFO-style (First In First Out) and can only store a fixed number of messages. The developer has to apply timed model checking to make sure the in-buffer does not overflow. Furthermore, we explicitly consider that a transmission of a message from sender to receiver takes time. Therefore, connectors have a required message

transmission delay. The delay is defined as an interval with a minimum and maximum bound. Please refer to [48] for a detailed specification of Real-Time Coordination Protocols.

Real-Time Statecharts

MechatronicUML defines the coordination behavior in a state-based manner using *Real-Time Statecharts* (RTSCs). They tackle typical aspects of interconnected mechatronic systems like hard real-time requirements and asynchronous message-based coordination. Their bases are hierarchical UML state machines and timed automata. Using Real-Time Statecharts, the developer defines the behavior of a role, a port, or a component. Like timed automata, a Real-Time Statechart may define a finite set of continuous clocks that enable clock-based timing requirements like guards, state invariants, and worst-case execution times. Each state may define state invariants and side effects. Each invariant defines an upper-bounded clock constraint that refers to a clock of the statechart. All invariants must be true as long as the state is active, otherwise a time stopping deadlock occurs. Timed Model-Checking is applied to identify these deadlocks. Please refer to [28] for a detailed description of Real-Time Statecharts and their execution semantics.

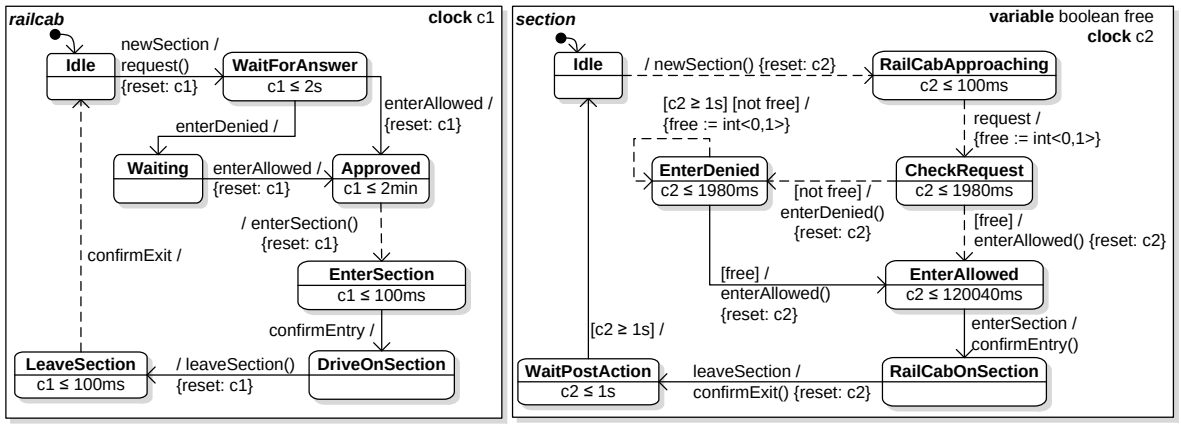

Figure 3.9: Real-time Statecharts of the Coordination Protocol "*EnterSection*" (from: [62])

Figure 3.9 shows the Real-Time Statecharts of the two roles `railcab` and `section` of the Real-Time Coordination Protocol `EnterSection` (cf. Fig. 3.8). As they belong to one protocol, they are able to exchange messages with each other. The informal behavior definition for these roles is as follows: Initially, both roles are in state `Idle`. As soon as role `section` recognizes the RailCab (this behavior is not part of the protocol because it shall be defined within the `section` component), it sends the message `newSection` to role `railcab` to inform that a new section is approaching. Afterwards, it switches to state RailCabApproaching. Here, it is waiting for a request for at most 10 ms. When role `railcab` receives and consumes the message, it requests for entering this section by answering with the message `request` and switching to state `WaitForAnswer`. According to the invariant, the answer must be available after 2 s. Role `section` receives and consumes message `request` at most 100 ms after this state was activated. Then, role `section` switches to state `CheckRequest`. In this state, the section checks within 1980 ms if the RailCab may enter. This check is not part of the protocol but of the concrete component because each section kind must execute different checks. However,

the result is stored in variable free and may be `true` or `false`. If the section is free, then the role `section` sends the message `enterAllowed` and switches to state `EnterAllowed`, else it sends the message `enterDenied` and switches to state `EnterDenied`. For the latter case, the role will check repeatedly, if the section is free. As soon as this is the case, it will send `enterAllowed` and switch to the eponymous state. As long as role `section` did not answer if entering is allowed, role `railcab` remains in state `WaitForAnswer`. However, after 2 s, it will either receive message `enterAllowed` or `enterDenied`. If entering is denied then it will wait until message `enterAllowed` arrives. For simplicity reasons, we assume that entering will eventually be allowed within the 1980 ms. As soon as this happens, role `railcab` switches to state `Approved`. As depicted by the state invariant, the approval is valid for 2 min. Within this time, the RailCab must enter the section. As soon as this is the case, message `enterSection` will be sent. Role `section` will receive this message at most 120,040 ms (= 2 min 40 ms) after it allowed to drive on its section and will answer with message `confirmEntry`. Here, the invariant of 2 min and 40 ms constitutes of the 2min the RailCab had to drive to the section and two times the message delay for the two messages `enterAllowed` and `enterSection`. Role `railcab` will consume message `confirmEntry` at most 100 ms after it sent message `enterSection`. As soon as the RailCab leaves the section, it will send message `leaveSection`. Role `section` will confirm this with the message `confirmExit`. After one second, the coordination for this drive through is finished and a new coordination with the same RailCab can start again. Role `railcab` will consume message `confirmExit` within 100 ms after message `leaveSection` was sent. Then, it is also ready to enter this section again.

Compositional Verification

In this section we give a summary of the compositional verification approach in MechatronicUML. The approach syntactically decomposes the software model of an interconnected mechatronic system into Real-Time Coordination Protocols and components. For deriving correct verification results, it imposes one critical syntactical requirement to the MechatronicUML model: Any connection between two ports in the component model needs to be specified by a Real-Time Coordination Protocol. Each port of a component needs to implement one of its roles. That enables to verify the software model in three steps as shown in Figure 3.10.

In the first step, each Real-Time Coordination Protocol is verified for safety and liveness properties (cf. Sect. 3 in [62]). The Real-Time Coordination Protocol is the *abstract protocol* in our approach. The safety and liveness properties are proved via timed model checking, utilizing Uppaal as the underlying model-checker. This is achieved by a semantics preserving transformation of the Real-Time Coordination Protocol and its Real-Time Statecharts into a network of timed automata. A detailed explanation of the transformation is given by Gerking in [56]. Figure 3.11 illustrates the transformation approach. For each role and its corresponding Real-Time Statechart, a set of timed automata is generated (among others one timed automaton for each hierarchy level) and all states and transitions are adapted. Afterwards, the language elements that Uppaal does not support are transformed into elements of Uppaal (e.g. asynchronous message exchange, state events, urgent transitions, and hierarchical states, cf. [56]). Along with the generated timed automata, the model checker requires as a second input safety and liveness properties, which shall be verified. Therefore, in a first step, the developer has to specify its requirements as safety and liveness properties by using the Timed Computation Tree Logic (TCTL). In our example, the Real-Time Coordination Protocol `EnterSection` has the following informal requirement: “The message `enterSection` will never be sent from role

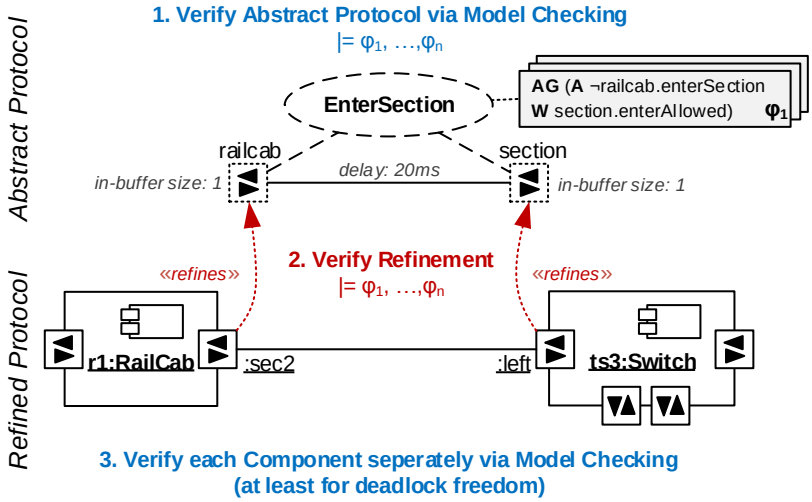

Figure 3.10: Overview of the Compositional Verification Approach (from: [62])

`railcab`, until message `enterAllowed` is sent by role `section`." The developer formalizes this into the TCTL property $\varphi 1$ (cf. Fig. 3.11):

```
AG (A not railcab.enterSectionMsg W section.enterAllowedMsg)
```

The developer manually transforms this property into the following UPPAAL TCTL expression:

```
A[] (railcab.enterSectionSent==true imply section.enterAllowedSent==true)
```

Here, `railcab` and `section` are the names of two timed automata. `enterSectionSent` and `enterAllowedSent` are two Boolean variables. Both are initially `false`. They are set to `true` when their respective message was sent and set to `false` when location `Idle` is activated. These variables are necessary because UPPAAL does not support the operator `W` and is not able to reference a message within its TCTL expression but only locations, variables, and clocks.

In the second step, one has to verify if the ports of the components refine the roles of the Real-Time Coordination Protocol correctly (cf. section 4 in [62]). The behavior of a port must be compliant to the specified role behavior, i.e. it must be a legal refinement regarding the chosen refinement definition (cf. section 5 in [62]). The result is the *refined protocol*. Typical refinement steps include adding data exchange between different ports of a component, adding component specific functions, and accessing shared variables inside the component. That, in turn, may require to add additional states and transitions to the Real-Time Statechart of the role. The behavior of the refined protocol is defined by the port Real-Time Statechart. Moreover, the specification of the message buffer for incoming messages and the message transmission delay must conform to the protocol specification. In our example in Fig. 3.10, the connection between the ports `sec2` and `left` must be compliant to the Real-Time Coordination Protocol `EnterSection`. In particular, port `sec2` must be compliant to role `railcab`; port `left` must be compliant to role `section`. Thus, one has to check if the two Real-Time Statecharts of Fig. 3.9 are refined correctly by their corresponding port Real-Time Statecharts. Right now, MECHATRONICUML supports six refinement definitions for interconnected mechatronic systems and their automatic selection: `Simulation`, `Bisimulation`, `Timed Ready Simulation`, `Timed Simulation`, `Relaxed Timed Bisimulation` and `Timed Bisimulation`. They differ in

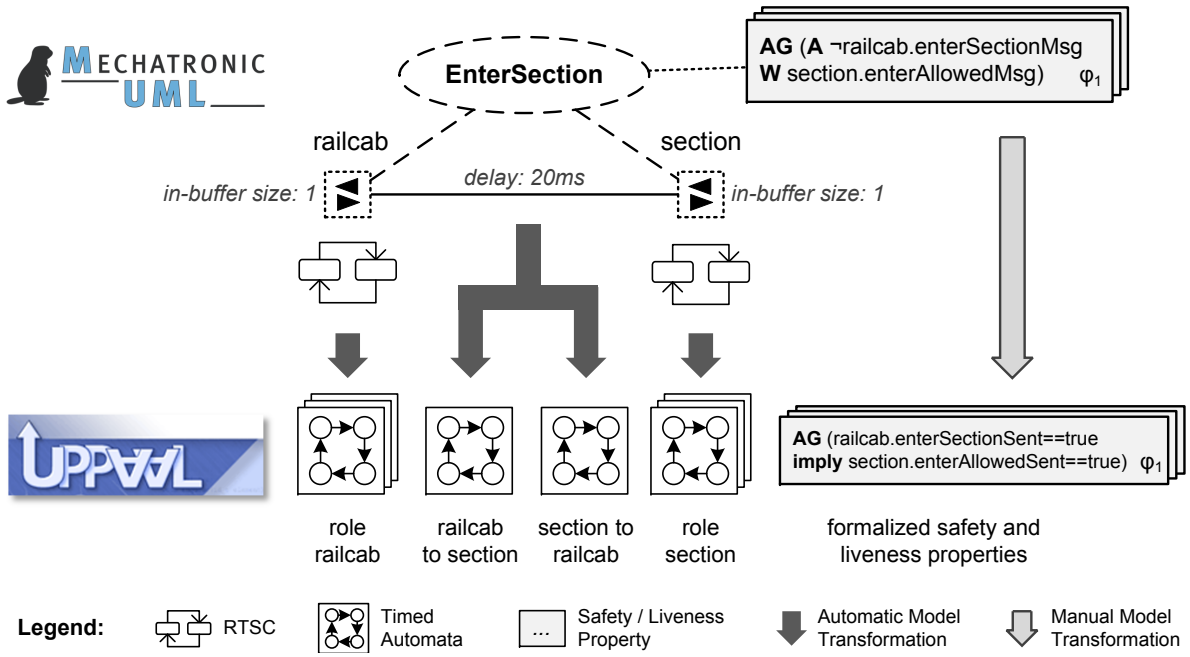

Figure 3.11: Transforming Real-Time Coordination Protocol EnterSection into UPPAAL timed automata (from: [62])

the properties that they preserve and modifications that they allow. A detailed description of this approach including the different refinement definitions is presented in sections 4,5,6 in [62].

In the third step, one need to verify for each component that it is free of deadlocks. Additional safety and liveness properties might be verified referring to a correct interaction of the different ports of a component if necessary. In our example, only one RailCab may receive permission to enter a track section. Such properties may only be verified on the component level because they depend on the interaction of the different ports of one component, e.g., the ports `left` and `right` of the `NormalTrackSection` in Fig. 3.7.

The overall compositional verification approach of MECHATRONICUML is still under active research. For example, current research activities focus on automating the synthesizing of the component implementations from the refined ports. This would complete step three of the presented approach. Furthermore, the MECHATRONICUML developers aim to transform the counterexamples generated by UPPAAL back to MECHATRONICUML language elements. This would make the usage of the attached model-checking tools more accessible to domain experts.

3.5 Test case generation tools and frameworks

Former described tools

- MBT-Tool,
- MaTeLo
- JUMBL
- Spec Explorer

can also be used for generating test cases. And there are a lot of tools available with the label "test case generation". Some of them are implementations of academic approaches, other with commercial background. One has to know about its background, its domain, its test infrastructure, etc. before deciding which tool fits best the own requirements.

3.5.1 Conformiq Designer

Conformiq Designer [2] is a classic Model Based Testing tool for automated test design. With this tool test cases can be generated when a model of the requirements (a specification model) is previously designed. Main application domain of Conformiq Designer is the telecommunication domain, but also other application domains like automotive, industrial automation are addressed.

Specification models can be created as UML State Machines and in Qtronic Modelling Language (QML). QML is a Java based language. Tests can be exported to test management tools or TTCN-3. It is a desktop product that runs on Windows and Linux. The test generation engine reads the models and runs a symbolic test generation algorithm that creates a set of tests to test the features described in the model. It generates automatically both test input data as well as test output data from the single model. These tests can be then reviewed and exported, but they cannot be executed as Conformiq Designer is not a test execution tool. They need to be exported in order for them to be useful. This means that they are written out in a human-readable or an executable format, for example Java, Python, Perl, HTML or Word-compatible formats. Many different output formats are supported.

3.5.2 T-VEC

T-VEC Tester for Simulink and Stateflow [13] generates test cases for MathWorks Simulink models based on various structural coverage criteria (structured path coverage, branch coverage, statement coverage or interface coverage). It automates much of the testing process by analysing the Simulink model to determine the best test cases for validating the model and testing implementations of the model.

The T-VEC Tester produces a complete set of artifacts for verifying Simulink models such as model analysis report for identifying model errors, test vectors (input values, expected output values, traceability from each test to the Simulink model), test coverage report or test results report that details test successes and failures.

3.6 Other verification tools and frameworks

3.6.1 Oracle Java Mission Control

Oracle Java Mission Control [3] is a tool suite for managing, monitoring, profiling and troubleshooting Java applications without introducing the performance overhead normally associated with these types of tools. It uses data collected for normal adaptive dynamic optimization of the Java Virtual Machine (JVM) and has been included in Java standard Java SDK since version 7u40.

Java Mission Control consists of the client application (JMC client) and a number of plug-ins that run on it:

- JVM Browser: shows the running Java applications and their JVMs. Each JVM instance is called a JVM Connector.
- JMX Console: connects to a running JVM, collects and displays its characteristics in real time, and enables you to change some of the runtime properties through Managed Beans (MBeans).
- Java Flight Recorder (JFR): collects and displays application characteristics for historical analysis and profiling.

Java Mission Control plug-ins connect to a JVM using the Java Management Extensions (JMX) agents. The JMX Console is a tool that presents live data about memory and CPU usage, garbage collections, thread activity, attributes exposed via MBeans and registered with the MBean server, and more features across the board, providing information about the Java VM.

The JMX Console also provides triggers that can monitor MBeans and trigger actions, such as showing a notification, when a condition is met.

The overview tab of JMX Console provides an overview of CPU performance and heap usage during the recording Figure 3.12.

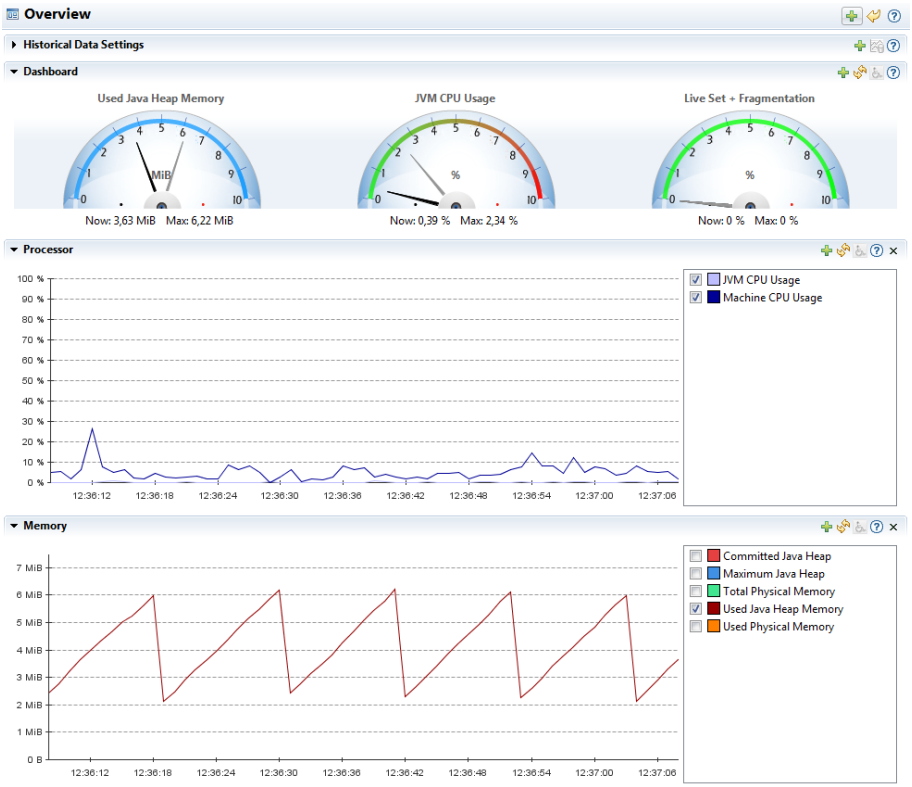

Figure 3.12: Java Mission Control Overview

Besides, the MBean tab displays information about all the Mbeans registered with the platform MBean Figure 3.13.

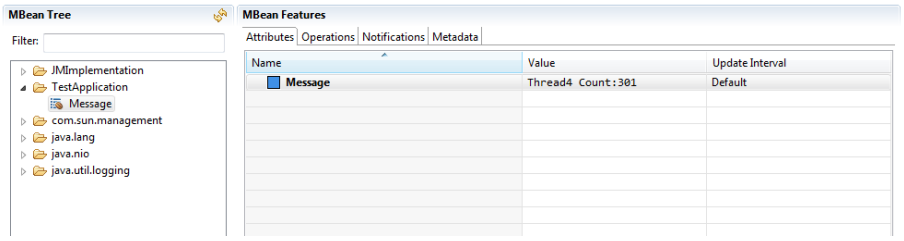

Figure 3.13: MBean Features

The main focus of this tool is on production-time profiling and diagnostics. Java Mission Control gathers the data necessary with the lowest possible impact on the running system. The technology used also enables the application to run at full speed once the tool is disconnected from the Java Virtual Machine (JVM). This makes JMC suitable for use in production environments.

3.7 Simulation Tools

3.7.1 TA Simulator

TA Simulator [104] is a discrete, event-based simulation tool as introduced in Section 2.7.2. It operates on the AUTOSAR-compliant and AMALTHEA-compliant Timing Model [98]. During simulation, state transitions (events) like the activation, preemption and termination of tasks as well as the respective time stamps are recorded into a BTF trace [24]. TA Simulator also includes an evaluation module which applies different statistical estimators to the trace, which allows the calculation of different timing metrics and performance metrics.

Figure 3.14 shows a screenshot of TA Simulator. In the upper part of the figure, the resulting trace of a simulation run is visualized in a Gantt chart. In addition to that, the interference analysis is shown as overlay to the Gantt chart. This overlay visualizes which tasks interfered with others (e.g. preemptions or resource conflicts). The middle part of the figure shows the load of the different cores over time. In the lower part of the figure, the results of the timing requirements analysis are shown. The left part hereby shows the different requirements and the respective percentage of requirement violations. In the right part, the response time histogram of a task together with its deadline (upper limit for the response time) is shown.

TA Simulator is integrated into the AMALTHEA Toolchain using the import/export interface of TA Tool Suite to transform an AMALTHEA model into a Timing Model and vice versa.

3.8 Trace Analysis and Verification Tools

3.8.1 Trace Compass

Trace Compass [49] is a new Eclipse project, which focuses on the development of a tool for viewing and analyzing any type of logs or traces. To be exact, it is actually not new, but was part of the Eclipse LTTng Project (Linux Trace Toolkit, next generation) before. This was due to the reason, that the capabilities of the developed tool were not just limited to traces from

Figure 3.14: Screenshot of TA Simulator (cf. [104]).

the Linux operating system, but could handle a large variety of different traces and formats instead.

In order to show that fact, the project developed an interface to the BTF trace format [24], whose specification was made open source in the scope of the AMALTHEA project. This functionality is available since their release version 1.0.0. Figure 3.15 shows a screenshot of a previous version of LTTng. As it can be clearly seen, the goal of Trace Compass is to provide a variety of views, graphs, and metrics to help analyzing traces by extracting and visually representing useful information.

It is developed to either be integrated into an existing Eclipse IDE or used as a standalone application. An available generic interface for the integration of logs or trace data input, analyses and views makes it possible to easily adopt new trace formats and extend the platform accordingly. Moreover, it supports live log and trace reading and monitoring and was especially designed to be able to handle traces that consume a huge amount of memory.

3.8.2 TA Inspector

TA Inspector [104] is a trace visualization, trace analysis and trace verification tool. It supports various trace sources, including Lauterbach, iSystem, Gliwa, Elektrobit, AMALTHEA and BTF traces.

An unique feature of TA Inspector is the support for a Closed-loop development process. Hereby, information from model-based descriptions like the AMALTHEA model, Code-Parsing and hardware traces are used to either create or refine Timing Models. Hardware traces are used in this context to automatically reconstruct execution times of runnables or the activation pattern of tasks.

Figure 3.16 shows a screenshot of TA Inspector. In the upper part of the figure, an imported trace is visualized in a Gantt chart. In addition to that, different event-chains are shown as

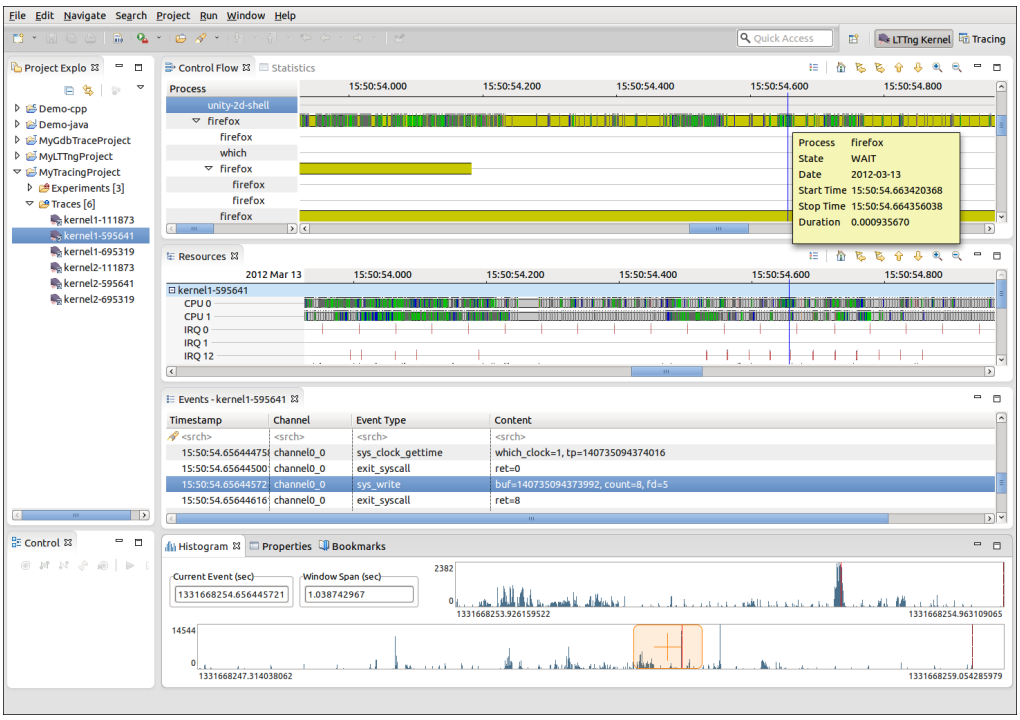

Figure 3.15: Screenshot of LTTng, the predecessor of Trace Compass (cf. [49]).

overlay to the Gantt chart. Event-Chains are defined by a stimulus and response event, e.g. the activation or termination of a task, and allow a detailed analysis of the duration of certain processing chains. The middle part of the figure shows the statistics of the event-chain analysis on the left and the histogram of an event-chain on the right. In the lower part of the figure, the results of a timing requirement analysis are shown.

TA Inspector is integrated into the AMALTHEA Toolchain using the import/export interface of TA Tool Suite to transform Timing Models resulting from the Closed-loop development process back to AMALTHEA models.

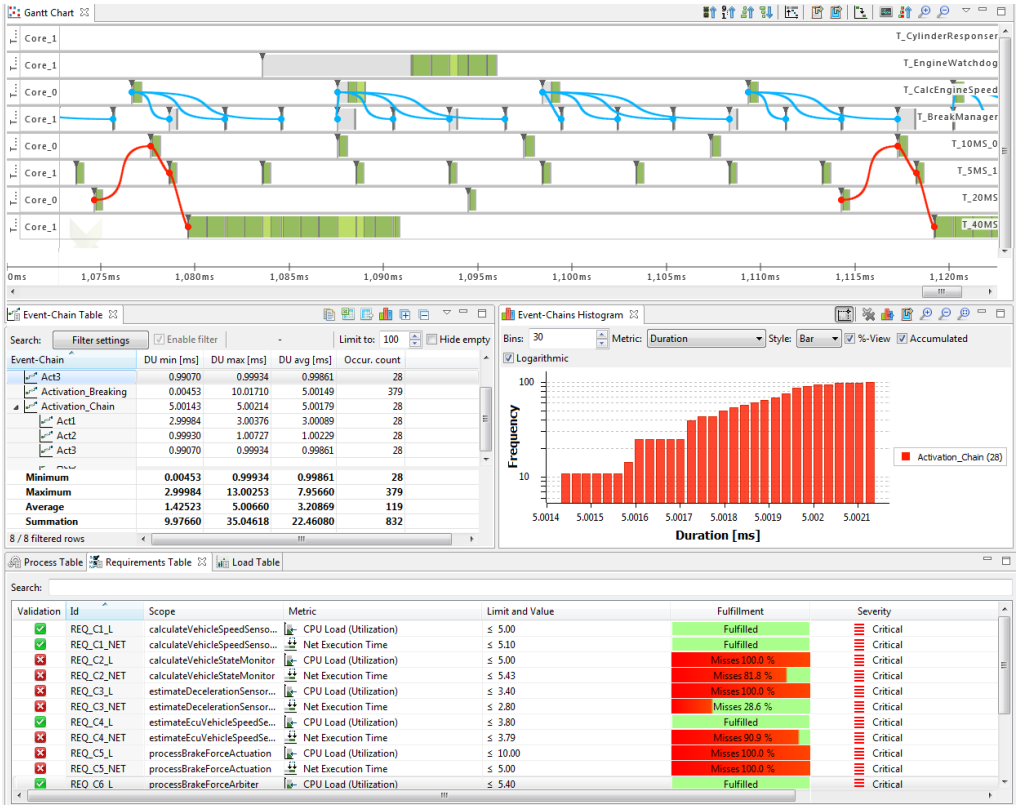

Figure 3.16: Screenshot of TA Inspector (cf. [104]).

4 Conclusion

The main contribution of this document is the analysis of the V&V techniques as well as a description of the main tools applied to model verification, in the context of the intended Amalthea4Public goals.

Based on this deliverable, the following step is to identify which techniques will be implemented inside Amalthea's toolchain and how it will be performed.

Bibliography

[1] Cleverbot. http://en.wikipedia.org/wiki/Cleverbot.

[2] conformiq designer. http://www.conformiq.com/products/conformiq-designer/.

[3] Java mission control. http://www.oracle.com/technetwork/java/javaseproducts/mission-control/index.html.

[4] Java pathfinder. http://en.wikipedia.org/wiki/Java_Pathfinder.

[5] Junit. http://junit.org/.

[6] Matelo. http://www.all4tec.net/index.php/All4tec/matelo-product.html.

[7] Mbt-tool. http://www.hjp-consulting.com/test-tools/model-based-testing-tools.

[8] Modeljunit. http://www.cs.waikato.ac.nz/~marku/mbt/modeljunit/.

[9] Nusmv. http://en.wikipedia.org/wiki/NuSMV.

[10] Spec explorer. http://research.microsoft.com/en-us/projects/specexplorer/.

[11] Spin. http://spinroot.com.

[12] Symbolic pathfinder. http://babelfish.arc.nasa.gov/trac/jpf/wiki/projects/jpf-symbc.

[13] T-vec. http://www.t-vec.com/solutions/simulink.php.

[14] Torx. http://fmt.cs.utwente.nl/tools/torx/introduction.html.

[15] Uppaal tron. http://people.cs.aau.dk/~marius/tron/.

[16] Zf set theory. http://en.wikipedia.org/wiki/Zermelo%E2%80%93Fraenkel_set_theory.

[17] A.F. Ackerman, P.J. Fowler, and R.G. Ebenau. Software inspections and the industrial production of software. 1983.

[18] W.R. Adrion, M.A. Branstad, and J.C. Cherniavsky. Validation, verification, and testing of computer software. 1982.

[19] S.A. Ajila and A.B. Kaba. Using traceability mechanisms to support software product line evolution. In *Information Reuse and Integration, 2004. IRI 2004. Proceedings of the 2004 IEEE International Conference on*, pages 157–162, Nov 2004.

[20] A Alhroob, K Dahal, and A. Hossain. Automatic test cases generation from software specifications. 2010.

[21] Muhammad Hamad Alizai, Lei Gao, Torsten Kempf, and Olaf Landsiedel. Tools and modeling approaches for simulating hardware and systems. In Klaus Wehrle, Mesut Güneş, and James Gross, editors, *Modeling and Tools for Network Simulation*, pages 99–119. Springer, 2010.

[22] S. Apel and C. Kästner. An overview of feature-oriented software development. *Journal of Object Technology*, 2009.

[23] J.D. Arthur, R.E. Nance, and S.M. Henry. A procedural approach to evaluating software development methodologies: The foundation. 1986.

[24] Eclipse AutoIWG. Btf specification v2.1.3. `https://wiki.eclipse.org/images/e/e6/TA_BTF_Specification_2.1.3_Eclipse_Auto_IWG.pdf`, 2014.

[25] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.

[26] Jerry Banks, John S. Carson, Barry L. Nelson, and David M. Nicol. *Discrete-Event System Simulation*. Prentice Hall, 2010.

[27] Don Batory. Feature models, grammars, and propositional formulas. In *Proceedings of the 9th International Conference on Software Product Lines*, SPLC'05, pages 7–20, Berlin, Heidelberg, 2005. Springer-Verlag.

[28] Steffen Becker, Stefan Dziwok, Christopher Gerking, Christian Heinzemann, Sebastian Thiele, Wilhelm Schäfer, Matthias Meyer, Uwe Pohlmann, Claudia Priesterjahn, and Matthias Tichy. The mechatronicuml design method - process and language for platform-independent modeling. Technical Report tr-ri-14-337, Heinz Nixdorf Institute, University of Paderborn, March 2014. Version 0.4.

[29] Beckert and Hähnle. Reasoning and verification: State of the art and current trends. 2014.

[30] G. Behrmann, A. David, K.G. Larsen, J. Hakansson, P. Petterson, Wang Yi, and M. Hendriks. Uppaal 4.0. In *Quantitative Evaluation of Systems, 2006. QEST 2006. Third International Conference on*, pages 125–126, Sept 2006.

[31] B. Beizer. Software testing techniques. 1990.

[32] David Benavides, Sergio Segura, and Antonio Ruiz-Cortes. Automated analysis of feature models 20 years later: A literature review. *Information Systems*, 35(6):615 – 636, 2010.

[33] David Benavides, Sergio Segura, Pablo Trinidad, and Antonio Ruiz-Cortes. Using java csp solvers in the automated analyses of feature models. In *Proceedings of the 2005 International Conference on Generative and Transformational Techniques in Software Engineering*, GTTSE'05, pages 399–408, Berlin, Heidelberg, 2006. Springer-Verlag.

[34] Shawn Anthony Bohner. *A Graph Traceability Approach for Software Change Impact Analysis*. PhD thesis, Fairfax, VA, USA, 1995. UMI Order No. GAX95-42995.

[35] C. Brink, P. Heisig, and S. Sachweh. Change impact analysis in system families. In *Software Engineering and Advanced Applications (SEAA), 2015 41th EUROMICRO Conference on*, Aug 2015.

[36] C. Brink, M. Peters, and S. Sachweh. Configuration of mechatronic multi product lines. In *Proceedings of the 3rd International Workshop on Variability & Composition*. ACM, 2012.

[37] M Chen, P Mishra, and D. Kalita. Coverage-driven automatic test generation for uml activity diagrams. 2008.

[38] Andreas Classen, Patrick Heymans, and Pierre-Yves Schobbens. What's in a feature: A requirements engineering perspective. In *Fundamental Approaches to Software Engineering*, volume 4961 of *Lecture Notes in Computer Science*, pages 16–30. Springer Berlin Heidelberg, 2008.

[39] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley Professional, 2001.

[40] K. Czarnecki and U. Eisenecker. *Generative Programming Methods, Tools, Applications*. Addison-Wesley, Boston, 2000.

[41] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. Staged Configuration Through Specialization and Multi-Level Configuration of Feature Models. In *Proceedings of the 3rd International Software Product Line Conference*, 2005.

[42] B Daniel, T Gvero, and D. Marinov. On test repair using symbolic execution. 2010.

[43] B Daniel, V Jagannath, D Dig, and D. Marinov. Fitness-guided path exploration in dynamic symbolic execution. 2009.

[44] Alexandre David, Gerd Behrmann, Peter Bulychev, Joakim Byg, Thomas Chatain, Kim G. Larsen, Paul Pettersson, Jacob Illum Rasmussen, Jiří Srba, Wang Yi, Kenneth Y. Joergensen, Didier Lime, Morgan Magnin, Olivier H. Roux, and Louis-Marie Traonouez. Tools for model-checking timed systems. In Olivier H. Roux and Claude Jard, editors, *Communicating Embedded Systems – Software and Design*, pages 165–225. ISTE Publishing / John Wiley, October 2009.

[45] M.S. Deutsch. Software verification and validation: Realistic project approaches. 1982.

[46] Jessica Diaz, Jennifer Perez, Juan Garbajosa, and AlexanderL. Wolf. Change impact analysis in product-line architectures. In Ivica Crnkovic, Volker Gruhn, and Matthias Book, editors, *Software Architecture*, volume 6903 of *Lecture Notes in Computer Science*, pages 114–129. Springer Berlin Heidelberg, 2011.

[47] J.H. Dobbins. Inspections as an up-front quality technique. 1987.

[48] Stefan Dziwok, Christopher Gerking, Steffen Becker, Sebastian Thiele, Christian Heinzemann, and Uwe Pohlmann. A tool suite for the model-driven software engineering of cyber-physical systems. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 715–718, New York, NY, USA, November 2014. ACM.

[49] Eclipse. Trace compass. `https://projects.eclipse.org/projects/tools.tracecompass`, 2015.

[50] R.E. Fairley. Tutorial: Static analysis and dynamic testing of computer software. 1975.

[51] R.E. Fairley. Tutorial: Static analysis and dynamic testing of computer software. 1976.

[52] R.E. Fairley. Tutorial: Static analysis and dynamic testing of computer software. 1978.

[53] Q Farooq, MZ Iqbal, ZI Malik, and A. Nadeem. An approach for selective state machine based regression testing. 2007.

[54] U Farooq and CP. Lam. Evolving the quality of a model based test suite. 2009.

[55] Briand L Garousi V and Labiche Y. Control flow analysis of uml 2.0 sequence diagrams. `http://www.sce.carleton.ca/squall/pubs/tech_report/TR_SCE-05-09.pdf`, 2011.

[56] Christopher Gerking. Transparent uppaal-based verification of mechatronicuml models. Master's thesis, Software Engineering Group, Heinz Nixdorf Institute, University of Paderborn, May 2013.

[57] S Gnesi, D Latella, M Massink, V Moruzzi, and I. Pisa. Formal test case generation for uml state charts. 2004.

[58] M.W. Godfrey and D.M. German. The past, present, and future of software evolution. In *Frontiers of Software Maintenance*, 2008.

[59] M. Gries. Methods for evaluating and covering the design space during early design development. *Integration, the VLSI Journal*, 38(2):131–183, December 2004.

[60] James Gross and Mesut Güneş. Introduction. In Klaus Wehrle, Mesut Güneş, and James Gross, editors, *Modeling and Tools for Network Simulation*, pages 1–11. Springer, 2010.

[61] M Harman, SG Kim, K Lakhotia, P McMinn, and S. Yoo. Optimizing for the number of tests generated in search based test data generation with an application to the oracle cost problem. 2010.

[62] Christian Heinzemann, Christian Brenner, Stefan Dziwok, and Wilhelm Schäfer. Automata-based refinement checking for real-time systems. *Computer Science - Research and Development*, pages 1–29, 2014.

[63] Christian Henke, Matthias Tichy, Tobias Schneider, Joachim Böcker, and Wilhelm Schäfer. Avec '08 organization and control of autonomous railway convoys.

[64] C.F. Hermann. Validation problems in games and simulations with special reference to models of international politics. 1967.

[65] CAR Hoare. An axionatic basis for computer programming. 1969.

[66] CP Hollocker. The standarization of software reviews and audits. 1987.

[67] J.E. Hopcroft and J.O. Ullman. An axiomatic basis for computer programming. 1969.

[68] R.L. Van Horn. Validation of simulation results. 1971.

[69] W.E. Howden. Functional program testing. 1980.

[70] CY Huang, JH Lo, SY Kuo, and MR. Lyu. Software reliability modeling and cost estimation incorporating testing-effort and efficiency. 1999.

[71] AZ Javed, PA Strooper, and GN Watson. Automated generation of test cases using model-driven architecture. 2007.

[72] K. C. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson. Feature-Oriented Domain Analysis (FODA), Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, 1990.

[73] Victor Khomenko. *Model Checking Based on Prefixes of Petri Net Unfoldings*. PhD thesis, Newcastle, GB, 2003. British Lending Library DSC stock location number: DXN061636.

[74] J.C. Knight and E.A. Myers. An improved inspection technique. 1993.

[75] B Korel, L Tahat, and B. Vaysburg. Model based regression test reduction using dependence analysis. 2002.

[76] N Kosindrdecha and J. Daengdej. A black-box test case generation method. 2010.

[77] C. Krueger. Eliminating the adoption barrier. *Software, IEEE*, 19(4):29–31, July 2002.

[78] Robert O. Lewis. Independent verification and validation. 1992.

[79] N Li, T Xie, N Tillmann, J Halleux, and W. Schulte. Reggae: Automated test generation for programs using complex regular expressions. 2009.

[80] Z. Liu, S. Magnus, J. Krause, and C. Diedrich. Concept for modelling and testing of individual mechatronic components for manufacturing plant simulation. In *Proceedings of Emerging Technology and Factory Automation (ETFA), 2014 IEEE*, ETFA '14, 2014.

[81] G.J. Myers. A controlled experiment in program testing and code walkthroughs/inspections. 1978.

[82] G.J. Myers. The art of software testing. 1979.

[83] USA Department of Defense. Defense Modeling and Simulation Office. "online m&s glossary." dod 5000.59m. 2009.

[84] Object Management Group (OMG). *Unified Modeling Language (UML) 2.4.1 Superstructure Specification*, August 2011. Document formal/2011-08-06.

[85] Paulius Paskevicius, Robertas Damasevicius, and Vytautas Stuikys. Change impact analysis of feature models. In Tomas Skersys, Rimantas Butleris, and Rita Butkiene, editors, *Information and Software Technologies*, volume 319 of *Communications in Computer and Information Science*, pages 108–122. Springer Berlin Heidelberg, 2012.

[86] Leonardo Passos, Krzysztof Czarnecki, Sven Apel, Andrzej Wąsowski, Christian Kästner, and Jianmei Guo. Feature-oriented software evolution. In *Proceedings of the Seventh International Workshop on Variability Modelling of Software-intensive Systems*, VaMoS '13, pages 17:1–17:8, New York, NY, USA, 2013. ACM.

[87] W Perry. Effective methods of software testing. page 26, 1995.

[88] W Perry. Effective methods of software testing. 1995.

[89] M. D. Petty. Modeling and simulation fundamentals: Theoretical underpinnings and practical domains. 2010.

[90] Uwe Pohlmann, Matthias Meyer, Andreas Dann, and Christopher Brink. Viewpoints and views in hardware platform modeling for safe deployment. In *Proceedings of the 2Nd Workshop on View-Based, Aspect-Oriented and Orthographic Software Modelling*, VAO '14, pages 23:23–23:30, New York, NY, USA, 2014. ACM.

[91] SJ Prowell. Jumbl: A tool for model-based statistical testing. 2003.

[92] C.V. Ramamoorthy, S.F. Ho, and W.T. Chen. On the automated generation of program test data. 1976.

[93] Stefan Rieger. Automatically generating test cases using uml structure diagrams. `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.170.1937`, 2011.

[94] P Samuel, R Mall, and P. Kanth. test case generation from uml communication diagrams. 2007.

[95] M Sarma, D Kundu, and R. Mall. Automatic test case generation from uml sequence diagrams. 2007.

[96] S.R. Schach. Software engineering. 1996.

[97] W. Schäfer and H. Wehrheim. The challenges of building advanced mechatronic systems. In *Future of Software Engineering*, 2007.

[98] S. Schmidhuber, M. Deubzer, R. Mader, M. Niemetz, and J. Mottok. Towards the derivation of guidelines for the deployment of real-time tasks on a multicore processor. In *Model-Based Safety and Assessment*, pages 152–165. Springer, 2014.

[99] L.W. Schruben. Establishing the credibility of simulations. 1980.

[100] I. Sommerville. Software engineering. 1996.

[101] SK Swain and DP. Mohapatra. Test case generation from behavioral uml models. 2010.

[102] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2002.

[103] N Tillmann and J. Pex de Halleux. White box test generation for .net. 2008.

[104] Timing-Architects. TA Tool Suite Version 15.01.0. TA Academic & Research License Program. [Online]. Available: http://www.timing-architects.com, 2015.

[105] A.M. Turing. Computing machinery and intelligence. 1963.

[106] Frank van der Linden. Software Product Families in Europe: The Esaps & Café Projects. *Software, IEEE*, 2002.

[107] Jim VanBuren and David A. Cook. Experiences in the adoption of requirements engineering technologies. 1998.

[108] A.E. Westley. Infotech state of the art report: software testing, volume 1: Analysis and bibliography. 1979.

[109] J. White, D. Benavides, D.C. Schmidt, P. Trinidad, B. Dougherty, and A. Ruiz-Cortes. Automated diagnosis of feature model configurations. *Journal of Systems and Software*, 83(7):1094 – 1107, 2010.

[110] R.B. Whitner and O. Balci. Guidelines for selecting and using simulation model verification techniques. 1989.

[111] Reinhard Wilhelm, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, Per Stenström, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, and Reinhold Heckmann. The worst-case execution-time problem-overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems*, 7(3):1–53, April 2008.

[112] T Xie, N Tillmann, J Halleux, and W. Schulte. Fitness-guided path exploration in dynamic symbolic execution. 2009.

[113] E. Yourdon. Structured walkthroughs. 1985.