Contributors:

Wen-ling Huang, BREU
Jan Peleska, BREU
Uwe Schulze, BREU

Editors:

John Fitzgerald, NCL
Peter Gorm Larsen, IHA

Reviewers:

Joey Coleman, AU
Adrian Larkham, Atego
Augusto Sampaio, UFPE,
Gustavo Henrique Porto de Carvalho, UFPE
## Document History

<table>
<thead>
<tr>
<th>Ver</th>
<th>Date</th>
<th>Author</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>25-11-2012</td>
<td>Jan Peleska</td>
<td>Initial document version</td>
</tr>
<tr>
<td>0.2</td>
<td>10-12-2012</td>
<td>Jan Peleska</td>
<td>Update for writing plan: completion of table of contents</td>
</tr>
<tr>
<td>0.3</td>
<td>11-01-2013</td>
<td>Jan Peleska</td>
<td>Abstract, Chapter 1 (Introduction), Part I and Part IV complete and ready for review (other parts still work in progress)</td>
</tr>
<tr>
<td>0.4</td>
<td>24-01-2013</td>
<td>Jan Peleska</td>
<td>First complete document version. Additional chapter on cross-domain tool qualification inserted in Part IV. Parts II and III completed. Chapter 11 still needs some improvements. Changes due to review remarks from Joey Coleman regarding Parts I, IV.</td>
</tr>
</tbody>
</table>
Abstract

This technical report describes the testing methodology provided for testing systems of systems modelled with SysML/CML. An introductory part explains the current state of the art and the challenges in SoS testing and indicates focus points of future work. Some of these challenges will be solved within the COMPASS project, as explained in the following parts of this document and in deliverable D34.2. The COMPASS testing methodology is based on the model-based testing paradigm, accepting SysML/CML test models as input. It is explained how the models are transformed into an abstract syntax tree internal to RT-Tester. Test cases are automatically identified by parsing the AST, and requirements tracing is performed by exploiting the relations between requirements and model elements that can be specified in SysML models. Transition relations are generated from an operational interpretation of (a subset of) SysML/CML models; they are used to calculate concrete test data, using constraint solving techniques for reachability goals similar to bounded model checking. The integration of RT-Tester into the COMPASS tool platform and protocols supporting this objective are specified. A generic concept for tool qualification is elaborated which conforms to ISO26262, RTCA DO-178C and CENELEC EN 50128-2011. This allows for applying the testing methodology and its supporting tool platform in a safety-critical SoS context.
5.2 Development Models Versus Test Models ........................................... 40
5.3 Model-Based SoS Testing ................................................................. 41
5.4 Basic MBT Automation Techniques .................................................. 42

6 CML/SysML Test Models – Supported Language Subsets .......................... 45
6.1 CML and SysML ............................................................................. 45
6.2 CML Subset Currently Supported for Operations and Datatypes .......... 49
   6.2.1 Types ......................................................................................... 49
   6.2.2 Expressions .............................................................................. 49
   6.2.3 Operations ................................................................................. 51
   6.2.4 Events ........................................................................................ 51
6.3 SysML subset .................................................................................. 51
   6.3.1 Packages ................................................................................... 52
   6.3.2 System Components ................................................................. 52
   6.3.3 Variables ................................................................................... 52
   6.3.4 State Machines ......................................................................... 53
   6.3.5 Operations ................................................................................. 53
   6.3.6 Data Types ............................................................................... 53
   6.3.7 Requirements .......................................................................... 54
   6.3.8 Constraints ............................................................................... 54
   6.3.9 Signals and Events ................................................................. 54
   6.3.10 Flow Ports and Item Flows .................................................... 54

7 Abstract Syntax Tree Representations for CML/SysML Test Models .......... 56
7.1 Generating the AST ........................................................................... 58
7.2 Example ............................................................................................ 58

8 Model-Based Test Case Identification ..................................................... 67
8.1 Computations, Traces and Model Coverage ...................................... 67
8.2 Model coverage strategies ................................................................. 68

9 User-Defined Test Cases and Their Formal Representation ................... 77
9.1 Test Case Specification Formalism .................................................... 77
9.2 SafetyLTL ....................................................................................... 78
9.3 Encoding SafetyLTL Formulas as BMC Instances ............................ 79

10 Traceability – from Requirements to Test Execution Logs ..................... 81
10.1 Test-Related Artefacts and Their Traceability Relationships ............... 81
10.2 Tracing Model Elements to Simple Requirements .............................. 82
10.3 Tracing Test Cases to Simple Requirements ...................................... 86
10.4 Tracing from the Model to Complex Requirements 86
10.5 Test Cases Tracing to Complex Requirements 87
10.6 User-Defined Test Cases 87

11 Generating the Transition Relation 88
11.1 Transition Relation Generation for SysML Test Models 89
11.1.1 Informal Description of SysML Semantics 89
11.1.2 Formalised Abstract Syntax 98
11.1.3 Symbolic Execution and Predicate Transformers 102
11.1.4 Transition Relation for SysML Models 107
11.2 Transition Relation for CML Interleaving Semantics 118

12 Automated Test Data Generation 124
12.1 SMT Solver 125
12.2 Abstract Interpreter 126

13 Test Model V&V By Bounded Model Checking 127
13.1 Bug Finding By Bounded Model Checking 127
13.2 BMC Against SafetyLTL Formulas 128

III COMPASS Platform Integration 129
14 Eclipse User Interface Plugin for RT-Tester 130
14.1 Architecture 130
14.2 RTT-MBT Java API 131
14.3 RTT-MBT Eclipse Plugin 134

15 The JSON Protocol for Interaction Between Tool Components 136
15.1 Structure 136
15.1.1 JSON Encapsulation 136
15.1.2 Command Sequence 136
15.2 Session management 137
15.2.1 Client Identification 137
15.2.2 Job Acknowledge 138
15.2.3 Testing Connections 138
15.2.4 Retrieve Status Information 138
15.2.5 Aborting Commands 139
15.2.6 Cleanup 139
15.3 Test Model 140
15.3.1 Storing Models 140
15.3.2 Checking Model Files ........................................ 140
15.3.3 Live-lock Checking ........................................ 141

15.4 Test Generation Context ...................................... 142
15.4.1 Configuration ............................................... 142
15.4.2 Signalmap .................................................. 142

15.5 Test Procedure Generation ................................... 143
15.5.1 Generating Test Procedures ............................... 143
15.5.2 Generating Simulations ................................... 144

15.6 Test Procedure Commands .................................... 145
15.6.1 Compilation ................................................ 145
15.6.2 Test Procedure Execution ................................. 145
15.6.3 Documentation ............................................ 146

15.7 Replay Test Results .......................................... 146
15.7.1 Replay ..................................................... 147

15.8 Rtt-Mbt Server Working Areas ............................... 147
15.8.1 Creating Work Areas ..................................... 148
15.8.2 Destroying Work Areas ................................... 149
15.8.3 Checking For A Work Area ............................... 149
15.8.4 Send Files To Work Areas ................................. 150
15.8.5 Check Files In Work Areas ............................... 150
15.8.6 Retrieve Files From Work Areas ......................... 151
15.8.7 Remove Files In Work Areas ............................. 153

15.9 User Feedback ................................................ 153
15.9.1 Progress Indication ...................................... 153
15.9.2 Console Output ........................................... 154

15.10 Debugging .................................................. 154
15.10.1 Error Messages ........................................ 154
15.10.2 Debug Information ..................................... 154
15.11 Compression ................................................ 155

16 Interaction with Artisan Studio .............................. 156
16.1 XMI Export .................................................. 156
16.2 CML Export ................................................. 157

17 Interaction with CML Tool Components ...................... 158
17.1 AST Transformation ......................................... 159
17.2 Coverage Information ...................................... 159
17.3 Bounded Model Checking .................................. 159
17.4 Exploiting Symbolic Test Cases Information ............ 160
IV RT-Tester Tool Qualification 161

18 Introduction to Tool Qualification 162
   18.1 Model-based Testing and Associated Tool Support . . . . . . 162
   18.2 Model-based Testing in Industry and Tool Qualification . . . 163
   18.3 Tool Qualification According to ISO 26262 . . . . . . . . . 164
   18.4 Contributions . . . . . . . . . . . . . . . . . . . . . . . . 166
   18.5 Outline of Part IV . . . . . . . . . . . . . . . . . . . . . . 166

19 Tool Classification According to ISO 26262 168
   19.1 Impact Analysis . . . . . . . . . . . . . . . . . . . . . . . . 168
   19.2 Tool Error Detection Capabilities . . . . . . . . . . . . . . . 169
   19.3 Tool Confidence Level . . . . . . . . . . . . . . . . . . . . . 170

20 Verification for Qualification 172
   20.1 Identification of Relevant Components . . . . . . . . . . . . 173
   20.2 Identification of Tool-internal Measures . . . . . . . . . . . 174
   20.3 Verification Activities . . . . . . . . . . . . . . . . . . . . . 174
   20.3.1 Test Strategy for the Parser Component . . . . . . . . . 177

21 Error Detection Using Replay 179

22 TQ-Related Requirements – Comparison Between ISO 26262,
   RTCA DO-178C and EN50128:2011 182
   22.1 Qualification of SoS V&V Tools . . . . . . . . . . . . . . . 182
   22.2 Characteristic TQ-Related Concepts . . . . . . . . . . . . . 183

23 Cross-Domain Tool Qualification Requirements for Model-
   Based SoS Testing 188

24 Related Work 192

25 Concluding Discussion 194

V Conclusion and Outlook 195

26 Conclusion 196

27 Outlook on COMPASS SoS Test Automation 198
# List of Figures

5.1 The model-based testing paradigm .................................................. 39
5.2 Development model versus test model as source for MBT .... 40
5.3 Test procedure generation using test case generator and SMT solver ................................................................. 43
6.1 Semantic models for CML/SysML ..................................................... 46
6.2 Test procedure generation from SysML models with optional CML operation and data type specifications .......................................................... 47
6.3 Test procedure generation from CML models ........................................ 48
6.4 Types of the CML expression language supported in the current RTT-MBT version 49
6.5 CML Expressions supported in the current RTT-MBT version 50
7.1 Two strategies to generate an AST from a SysML model ........ 57
7.2 The internal block diagram of SYSTEM ........................................... 58
7.3 The internal block diagram of SYSTEM ........................................... 59
7.4 The package view of SYSTEM in Artisan Studio ......................... 60
7.5 The internal block diagram of SystemUnderTest ......................... 61
7.6 The state machine FLASH_CTRL .................................................. 62
7.7 The substate machine EMER_ON ................................................. 63
7.8 The AST for the system component hierarchy .............................. 63
7.9 The AST for the state machine FLASH_CTRL ................................. 64
7.10 The AST for the state machine EMER_ON ..................................... 65
7.11 The hierarchy of events ............................................................. 66
7.12 Triggers ...................................................................................... 66
8.1 Behavioural model coverage strategies for MBT, I ............. 69
8.2 Behavioural model coverage strategies for MBT, II .................. 70
8.3 Structural coverage strategies for MBT ........................................... 75
10.1 Tracing model elements to requirements ....................................... 83
10.2 Model-based test cases tracing to the requirements shown in Fig. 10.1 ........................................ 85
11.1 Triggers for state machine transitions ................................................................. 91
11.2 Do activities supported by RTT-MBT and their equivalent representation by self loops .................................................. 97
11.3 Strongest post-condition predicate transformer for sequential behaviours .................. 105
11.4 CML specification RegisterProc process from [WCC+12] ............................................ 119
11.5 Test scenario with environment TE and system under test REG ......................... 120
11.6 Instrumented version of RegisterProc process ..................................................... 121
11.7 The initial condition $I$ associated with RegisterProc process ............................. 122
11.8 The transition relation $\Phi$ associated with RegisterProc process ..................... 123
13.1 Lasso-shaped path ................................................................. 128
14.1 Structure of the RTT-MBT Eclipse plugin ......................................................... 131
14.2 Overview of the RTT-MBT Java client API classes ............................................. 133
14.3 The RTT-MBT plugin classes .............................................................................. 135
19.1 Relation between tool impact (TI), tool error detection class (TD), tool confidence level (TCL), and automotive safety integrity level (ASIL) of the SUT ........................................ 171
20.1 High-level architecture of RTT-MBT ................................................................. 172
20.2 A simple state machine ......................................................................................... 177
20.3 IMR of the state machine in Fig. 20.2 ................................................................. 178
List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.1</td>
<td>Traceability matrix for Example 1</td>
<td>86</td>
</tr>
<tr>
<td>20.1</td>
<td>Functionality in RTT-MBT affected by tool-internal measures for error detection</td>
<td>173</td>
</tr>
<tr>
<td>20.2</td>
<td>Verification activities for affected components</td>
<td>175</td>
</tr>
<tr>
<td>22.1</td>
<td>Comparison between tool impact classification in ISO26262, RTCA DO-178C and EN50128:2011</td>
<td>184</td>
</tr>
<tr>
<td>23.1</td>
<td>Tool impact identification for MBT tools according to the three standards</td>
<td>189</td>
</tr>
<tr>
<td>23.2</td>
<td>Tool error detection level for MBT tools according to the three standards</td>
<td>189</td>
</tr>
<tr>
<td>23.3</td>
<td>Tool qualification levels for MBT tools like RT-Tester</td>
<td>189</td>
</tr>
<tr>
<td>23.4</td>
<td>Tool qualification requirements for MBT tools like RT-Tester, when applied to SoS testing in the automotive, avionic and railway domains</td>
<td>191</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction and Overview

SoS Testing – An Assessment. The size and complexity of systems of systems (SoS) prevents us from applying existing testing methods, tools and management practices successfully used on the component level in a direct way, because these methods do not scale up to the SoS level in a straight-forward way. This situation is analysed in Part I of this document, where

- the current state of practice for SoS testing (Chapter 2),
- the resulting challenges (Chapter 3), and
- novel scientific approaches to SoS testing (Chapter 4)

are assessed both from a technical and a managerial point of view.

Model-Based SoS Testing in COMPASS. Based on CML/SysML – the COMPASS core approaches to SoS modelling – this technical note introduces an approach to model-based testing (MBT) of SoS, together with its supporting tool platform. This is described in Part II. Chapter II introduces into the MBT paradigm. The CML/SysML subset currently supported for MBT is described in Chapter 6. These models are internally represented by abstract syntax trees (AST) (Chapter 7), so that algorithms for identifying test cases (Chapters 8, 9) and for determining the models’ operational semantics (Chapter 11) can run as visitors traversing the AST.

CML/SysML supports requirements engineering and tracing from model elements to requirements as an inherent part of the modelling formalism. This is exploited in our test automation approach to realise automated tracing between
• requirements,
• test cases,
• test procedures running these test cases, and
• test results,

as described in Chapter 10. While “relevant” test cases can be identified as logical constraints simply by traversing the AST, concrete test data has to be calculated by solving these constraints in a way that is consistent with the models’ transition relation. The logical framework for this approach and its practical realisation based on SMT solving is sketched in Chapter 12. The high degree of automation achieved by this MBT methodology and the associated tool platform increases the dependency on test model completeness and correctness. To this end, the COMPASS testing platform offers integrated functions for bounded model checking, so that SoS test models can be verified by exploring the vicinity of model states reachable by a bounded number of transitions (Chapter 13).

While many of the results presented here apply to MBT in general, the automated test case identification strategies described in Chapter 8 and 9 address problems and provide solutions which are specific to SoS testing.

**COMPASS Platform Integration.** In Part II of this document the integration of the MBT technology described in Part II into the COMPASS tool platform is described. The implementation of the MBT technology is realised by extending the RT-Tester tool developed by Verified Systems International GmbH in cooperation with the University of Bremen.

• The Eclipse user interface plugin for the COMPASS tool platform is described in Chapter 14.

• A tool communication protocol based on the JSON specification has been designed (Chapter 15). While this is currently used only for communication between client and server components of the test tool, this protocol can be applied and extended at a later stage for exchanging data between COMPASS tools in general.

• In Chapter 16 the interaction between RT-Tester and Artisan Studio – Atego’s tool supporting the development of SysML models – is described.

• Interaction between RT-Tester and the other COMPASS tools supporting CML is sketched in Chapter 17.
**MBT Tool Qualification.** Tools automating development or V&V activities for safety-critical systems have to be qualified. MBT tools such as the one used in COMPASS are rather complex and have code bases of considerable size. It is therefore desirable to elaborate a “light-weight” approach to MBT tool qualification (TQ) which can be easily repeated when developing tool increments. This is described in Part IV where such a light-weight approach to RT-Tester TQ is elaborated. This approach conforms to the ISO26262 standard which is applicable in the automotive domain (Chapters 18 to 21).

In the context of SoS, TQ raises an additional challenge, because the regulations for tool qualification are domain-specific, whereas SoS typically involve constituent systems from several domains. As a consequence, tools to be used across domains for SoS development and V&V need to comply with the rules imposed by all applicable standards. We exemplify this problem by analysing the differences between TQ requirements in the automotive, avionic and railway domains. This is described in Chapter 22 and the resulting overall criteria for tool qualification of MBT tools are captured.

**Conclusion and Outlook.** In Part V we present a conclusion about the MBT-related results achieved so far within the COMPASS project (Chapter 26) and give an outlook about the next COMPASS research objectives regarding model-based SoS testing (Chapter 27).
Part I

SoS Testing – State of Practice, Challenges and Focal Points of Research
Chapter 2

State of Practice

2.1 Compositionality Versus SoS System-Level Testing

Let us first consider why SoS testing at SoS-level is necessary at all, since – in theory – it should be possible to infer the SoS behaviour from that of its constituent systems: the concept of compositionality has been elaborated in the field of formal methods. Under certain conditions it allows the deduction of emergent functional properties \( P \) of a system \( S \) from the local properties \( P_i \) of its constituent systems \( S_i, i = 1, \ldots, n \). In its simplest form, for example, a distributed system \( S = (S_1 \parallel S_2 \parallel \ldots \parallel S_n) \) satisfies any specification \( P \) implied by the conjunction of its constituent systems’ properties, that is, \( S \text{ sat } P \text{ and } \bigwedge_{i=1}^{n} P_i \Rightarrow P \text{ implies } S \text{ sat } P \) [Hoa78]. In the light of the considerable costs caused by SoS integration testing it seems reasonable to investigate whether emergent properties \( P \) could not be simply deduced from local properties \( P_i \) by the way of compositional reasoning. Indeed, the distribution and local autonomy of constituent systems facilitate the application of compositional arguments, because prerequisites like the absence of shared resources (e.g. variables, processors) needed to apply such an argument are generally fulfilled.

These considerations obviously contradict practical experience with SoS testing on system level, where it quite frequently turns out that the composed system does not fulfil its expected emergent properties. This experience is not caused by the preconditions for the compositionality argument being violated. It is caused instead by
1. crucial non-functional emergent properties – in particular, safety and security [Lev95] – being non-compositional (that is, the combination of safety or security mechanisms does not necessarily lead to a safe or secure system);

2. insufficient quality of its constituent systems at the point in time when they are first delivered for the purpose of integration testing (that is, some $S_i$ satisfy $P'_i$ instead of $P_i$);

3. hidden undocumented assumptions made by the sub-system suppliers (some $S_i$ satisfy $(A_i \Rightarrow P_i \land P''_i) \land (\neg A_i \Rightarrow P'''_i)$ instead of $P_i$); and

4. emergent properties being insufficiently captured (that is, the true required emergent properties are $P \land Q$, while only $P$ has been documented).

It is the purpose of SoS system-level testing to reveal these deviations.

2.2 Coordination of Testing Activities

Using the terms of the military domain, testing activities of constituent systems may be structured into developmental test and evaluation (DT&E) which is a verification activity, and operational test and evaluation (OT&E) which is a validation activity [Dep08, p. 43]. As emphasised in [Dep08, p. 11] and [CCT08], SoS testing activities consist of separate concurrent and orthogonal threads. Since the life cycles of constituent systems are typically managed independently, “local” DT&E and OT&E activities are driven by the need to verify updates and extensions developed due to demands specific to the constituent system. A constituent system, however, may play a role in more than one SoS, and therefore the demand for improvements, changes and extensions may arise either from one or from several SoS in need of modified or new emergent capabilities. In the former case, testing activities will generally not be coordinated with test campaigns for all SoS the constituent system is a part of. In the latter case, the synchronisation points between concurrent testing activities performed on constituent system level are interoperability test campaigns. Interoperability testing verifies two or more constituent systems

1. to ensure their basic capabilities to exchange data over the intended interfaces, and,
2. to verify that the synergetic properties expected from this cooperation are realised in conformance with the SoS requirements.

The latter V&V activity is called *end-to-end* testing, since SoS functionality is investigated along the complete processing chain, from the initiating constituent system to the systems supporting and finally to those utilising the established results.

In a systematically structured SoS test campaign it is first ensured that constituent-level tests have been successfully completed, so that it may be expected that these systems comply with their specifications. In particular, it should be ensured that *conformance tests* have already been performed in order to verify that each constituent system conforms to the applicable (communication and/or functional) standards. Moreover, acceptance testing of constituent systems should already have ensured that system-specific functional, structural and non-functional properties are fulfilled. In the light of the discussion about compositionality above, this means that constituent-level testing activities ensure local properties \( S_i \) sat \( P_i \), \( i = 1, \ldots, n \), while interoperability testing investigates whether \( (S_1 \parallel S_2 \parallel \ldots \parallel S_n) \) sat \( P \) is fulfilled, that is, whether the expected emergent properties \( P \) are really ensured by the SoS.

It should be emphasised that interoperability testing is an accepted technique for testing distributed systems in general. For SoS, however, they are of particular importance since the cooperating components are less tightly integrated than, for example, the avionic controllers in an aircraft.

In many cases inter-operability testing relies on simulations. When following an integration test strategy incorporating only limited subsets of constituent systems at a time \([Dep08, \text{Fig. 4.3}]\) other SoS components may be needed to support the workflow required for the configuration under test. These other components may not be available as original equipment but have to be represented by simulations considered as part of the test equipment \([Dep08, \text{p. 70}]\). Moreover, simulations are needed in situations where (parts of) constituent systems are not yet available in order to allow for a comprehensive SoS system level test \([Dep08, \text{p. 80}]\).

## 2.3 Coping with complexity

The authors of \([LR09]\) address the combinatorial explosion problem caused by the size of SoS state and input vectors by adopting the concepts of
pairwise testing and orthogonal arrays for SoS system-level testing. These concepts have been widely applied in software testing. Pairwise testing with orthogonal arrays advocates test data generation according to the following strategy [Tat87], which goes back to Taguchi’s original ideas on robust design [Tag87, Pha89]:

1. Identify the input and state parameters influencing the System Under Test (SUT) behaviour (these parameters are called factors in the Taguchi Method).

2. Partition each single input or state vector component into equivalence classes (called levels in the Taguchi Method).

3. The orthogonal array approach is a method for selecting parameter-level combinations which are balanced in the sense that all combinations of a given dimension (n-tuples of levels associated with n factors) occur an equal number of times.

Typically, the orthogonal array method is applied to pairs of different factors, so that for each pair the associated levels are exhaustively combined, and each pair is exercised the same number of times (n = 2).

Pairwise testing with orthogonal arrays has been criticised to be overrated, mainly because in some experiments its error detection capabilities were not better than random testing. From the experimental results presented by [BS04] it appears that this only applies for smaller sizes of state and input vectors, where the probability to “hit” the right pairwise combinations just by means of random data generation is still high. For the typical vector dimensions to be expected in SoS testing, however, this probability decreases in a considerable way. As a consequence it can be expected that systematic construction using the orthogonal array method will lead to a better randomisation of test vectors than just generating random vector data.

Liang [LR09] evaluated their pairwise testing strategy with tests of an automated negotiation engine for a web-based SoS. The authors claim that they achieved good error detection rates with a low number of test cases. The SUT appears to be transaction oriented in a way that allows for assessing the correctness directly at the end-of-transaction (EOT) point in time. This technique seems to be well-suited for SoS where (1) every test case can be immediately triggered by means of a suitable input vector, (2) the internal system state can be pre-set for test purposes, and (3) the outcome of the transaction can be immediately assessed at EOT, for example, by investigating the response vector and the contents of the database. It has to be noted, however,
that these tests would not detect dependability issues such as insufficient reliability due to memory leaks. Therefore they have to be complemented by robustness tests investigating system behaviour over a longer duration.

As may be expected, pairwise testing does not solve all complexity issues in SoS testing, as will be discussed in more detail in Section 3.1.

Summarising the discussion above in the light of SoS characteristics, autonomy as well as independence of constituent systems is reflected by independent developmental and operational tests. Conformance and interoperability testing addresses the distributed nature of SoS. Evolution of constituent system functions and emergent properties are addressed by coordinated constituent-level and SoS-level testing campaigns, as long as the novel or updated constituent system functionality affects the SoS under consideration. Emergent SoS behaviour is checked by means of end-to-end tests.
Chapter 3

Challenges in SoS Testing

3.1 Complexity Considerations and Their Implications

When testing electronic circuits or software, the quality of a test suite is usually assessed by its capability to uncover (certain classes of) errors present in the SUT. In particular, exhaustive test suites are able to uncover every deviation of the SUT from its expected behaviour. The following complexity considerations show that it will not be possible to apply analogous assessment criteria to SoS test suites.

When analysing the complexity issues of SoS test case generation it is useful to distinguish between

1. transaction-based systems that allow to pre-set the state in an explicit way (for example, if their state space is managed by database systems whose contents may be manipulated for test purposes), and

2. reactive control systems whose responses depend on internal control states that cannot be directly set for test purposes, but only indirectly by means of the timed sequence of input vectors passed from test environment to the SUT.

In the former case, test cases may be specified by means of a single pair of input and pre-state vectors \((\vec{x}, \vec{s}_0)\), and the expected results can be checked by analysing the output and post-state vectors \((\vec{y}, \vec{s}_1)\) at end of transaction. This leads to a combinatorial testing problem of up to \(2^n+m\) test cases, where \(n\) and \(m\) are the bit-based dimensions of input and state vectors, respectively.
This complexity value is based on the assumption that the SoS really is combinatorial, that is, that there are no hidden “time bombs” such as memory leaks, overflowing counters or malicious failure injections that can only be discovered after certain sequences of transaction executions.

In the latter case the pre-state required for checking a given test objective must be reached by means of a timed input sequence \((t_i, \vec{x}_i), i = 1, \ldots, k\) driving the SUT into a state meeting the pre-conditions of the test case. Based on complexity considerations of Chow \[\text{Cho78}\] regarding the number of test sequences required to perform exhaustive tests of Mealy state machines, the effort asymptotically increases to \(2^{n+2m}\) test cases. In presence of timing conditions, \(n\) and \(m\) are not only determined by the size of the input and state domain, but also depend on the number of clocks used in the system and the resulting clock regions \[\text{SVD01}\].

Since \(n\) and \(m\) are the sizes of the bit vectors needed to encode input and state vectors as well as clock regions of the complete SoS, the numbers \(2^{n+m}\) or \(2^{n+2m}\) of tests required for exhaustive test suites will be so large that exhaustive testing by complete exploration of inputs, states and timing conditions is certainly infeasible. Even the indication of the percentage of an exhaustive set of test cases performed will be meaningless, since it will always remain a small fraction of one percent.

As a consequence the criteria for “sufficient” test coverage and the general assessment of SoS maturity achieved can never be based simply on the percentage of tests belonging to an exhaustive test suite performed. This implies that a completely different concept of “meaningful” test cases has to be developed for SoS. Moreover, it cannot be expected that any test strategy applied successfully for component testing will scale up so that similar error detection capabilities can be expected for SoS system testing. Error guessing or search-based testing methods \[\text{SLS06, AIB10}\], for example, are inapplicable because the probability to find an “interesting” SoS execution trace by random searches or genetic algorithms becomes infinitesimally small with increasing number of cooperating concurrent components, size of input and state vectors, length of test traces involved and timing conditions to be observed.

3.2 Management Issues

The challenge for SoS testing campaigns becomes visible on the level of system integration testing. \[\text{Sle06}\] identifies SoS-specific test management problems.
While the author’s analyses were performed in the context of the United States Army Strategic Software Improvement Program, we find her assessment to be significant for SoS testing in general.

According to the author, a central problem lies in the fact that – due to the multi-stakeholder situation prevalent in typical SoS developments – SoS system-level integration test campaigns lack clear management authority. As a consequence test strategies may be sub-optimal, and this may even contribute to insufficient SoS quality, since the strategies may fail to test the most critical emergent properties to be performed by the SoS. According to the author’s hypothesis the root-cause of this problem is the lack of a so-called Overarching Executive, a person with a clear understanding of the SoS capabilities to be validated, a mandate assigning sufficient authority to this person, together with sufficient funding to put corrective actions and improvement measures into practice [Sle06, p. 12]. The overarching executive contrasts with constituent systems experts, who frequently lack sufficient knowledge about the emergent SoS properties. The most severe consequences of this deficiency are identified as follows (these observations are also confirmed by [HV10]).

1. SoS requirements, that is, the emergent properties of SoS, are insufficiently specified, with the effect that SoS-level test cases are hard to identify and not trustworthy with respect to expected results definitions.

2. SoS system-level test campaigns are hard to manage, due to the multi-stakeholder problem preventing the elicitation of test facility and test campaign managers endowed with sufficient authority and funding. This results in incomplete information about system maturity and test campaign status in general.

3. SoS system-level test campaigns are hampered by insufficient quality of component systems delivered for system integration testing.

[CCT08] find the main cause for insufficient specification of emergent properties in the fact that the view on SoS is still too much focused on the capabilities of the constituent systems, instead of developing a net-centric view on SoS-level capabilities.

It is obvious that sub-system malfunctions should be detected already by the suppliers. Our interpretation of observation (3) above is that the different quality standards discussed below contribute to this situation. In contrast to this, for example, civil aircraft manufacturers acting as system integrators can expect that software-based sub-systems delivered for integration comply with the requirements of one standard (in this case RTCA DO-178B or RTCA
DO-178C [WG-92, WG-11]) which implies that a comprehensive verification and test suite – its thoroughness depending on the component’s criticality – has been exercised before delivery.

The management-related testing deficiencies identified above induce the risk of premature SoS entry into service, and the threat of safety and security-related malfunctions to occur during the system’s operational phase.

Just as SoS complexity prevents certain testing techniques to be applied, management considerations related to costs and effort forbid other techniques which have been proven to be useful on component level: for example, expert-driven manual exploratory testing [KFN99] is too expensive to be performed on SoS level, since there is no guarantee that “interesting” SoS behaviour will be uncovered and no systematic evidence is created during the exploration process which could be used for safety cases or certification evidence.

3.3 Impact of standards

The proliferation of standards – in [She97, She01] the term “framework quagmire” was coined to emphasise this rapidly increasing evolution – becomes particularly visible in the field of SoS: due to multiple stakeholders, multiple application domains being integrated in one SoS, and potential dual-use of components, a variety of standards will be applicable to SoS development, verification and validation, and certification. This has the following impact on testing SoS.

1. The completeness criteria for testing constituent systems will differ because they are tested according to different standards.

2. The completeness criteria for system-level testing of SoS will have to be based on a harmonised version of the test requirements defined in the different standards applicable to the constituent systems.

3. The completeness criteria for system-level testing of SoS will have to be further refined and extended in order to take into account the emergent functional properties and the non-functional characteristics of the complete SoS which goes beyond the properties of each isolated constituent system.

4. Analogous considerations apply to the testing methods to be applied to SoS system-level testing.
5. Test automation tools will often be applicable to different SoS-components and on SoS system-level testing itself. Tool qualification criteria, however, will differ for different constituent systems and on SoS system level.

To illustrate this situation, consider software-based systems for military applications which will typically be required to conform to the standards [Pub97, Pub06]. If, for example, an aircraft type to be applied for an SoS constituent system is intended for dual use, its avionic systems have to be certified according to [WG-92] or [WG-11] as well. If road vehicles intended for dual use should also be part of the same SoS, this affects the test tool qualification requirements which, as [BPS12] point out, differ in avionic standards [WG-92, WG-11] from the automotive standard [ISO09].

The impact of standards not only becomes visible for product-specific V&V, but also for the quality systems implemented by manufacturers contributing to SoS developments. While the ISO 9000 standards seem to be generally accepted as a minimal pre-condition for suppliers, several application domains have their specific refinements of the general quality-related regulations. In the domain of civil avionics, for example, suppliers contributing to airborne systems are required by Airbus, Boeing, GEAE and Rolls-Royce to be certified according to AS9100 [G-109], whereas the IRIS International Railway Industry Standard [IRI12] is required in the railway domain in Europe. In the automotive domain processes are regulated and assessed according to the Automotive SPICE model [Aut10b, Aut10a].

Since SoS are typically unique one-of-a-kind systems this impact of the framework quagmire is unlikely to improve in the future. Standards are usually elaborated for domains where many products of a similar kind are developed, so that application of a standard helps to increase product quality not only in a single development effort, but over several product generations of different manufacturers. This situation is less critical where SoSs have constituent systems from a single domain. Consider, for example, the European railway network. Currently there are only few railway lines crossing borders between European countries, but – due to political decisions of the European Commission – this situation is bound to be improved in the future. As a result, a European railway SoS will emerge, where the national railway authorities are the prominent stakeholders. Standardisation of electronic software-based system development is given by the so-called CENELEC standards [CEN93, EN506, CEN01a, CEN01b] which have been accepted – with some national extensions – by all European partners. The main capabilities of the European Train Control System have already been standardised
D34.1 - Test Automation Support (Public)

A future version of the CENELEC standards could address SoS-related V&V measures required to test the overall system. Since the stakeholders already agreed to accept the current version of those standards they are likely to agree on a future version containing regulations for V&V on SoS level.

3.4 Integration Test Environments

While [Sle06, p. 14] and [CCT08] take the existence of a centralised SoS integration and test facility for granted, [GPGG08] challenge this very concept by sketching a maritime SoS scenario and arguing that it would be far too time-consuming and costly to perform SoS acceptance testing in a non-operational testing environment. Instead, SoS system-level testing should be executed concurrently to the operation of its constituent systems. The authors justify this acceptance process at runtime by reasoning that run-time tests have to be performed in any case because of the dynamic nature of SoS: constituent systems joining or leaving the SoS configuration require runtime tests checking whether the SoS is still operable after these dynamic reconfiguration processes. Therefore it would be more effective to completely give up the idea of centralised test facilities.

It should be emphasised that while these considerations apply to the SoS level, dynamic reconfiguration and an acceptance process at runtime may be unacceptable on the level of constituent systems. Consider, for example, an SoS scenario from the civil transportation domain, involving trains, aircrafts and road vehicles such as buses. For the railway constituent system, the same considerations apply as for the complete SoS: new train or interlocking technologies are tested within the operational railway network. Test isolation – that is, the segregation between tested and operational components of the railway network – is ensured by exercising the new system components on sub-networks separated from the operational portions by means of point positions and train protection mechanisms. The railway network is dynamically modified and extended while being operational. In contrast to this, dynamic reconfigurations of an aircraft (e.g. another controller taking over the tasks of a failed one) is limited to statically (mostly hot standby) redundant components. The possibility of more dynamic re-configurations among avionic controllers of similar capabilities has only recently been discussed and has been limited to situations where the aircraft is still on ground. Dynamic reconfigurations for the purpose of extending the capabilities of an aircraft
during its actual operation are currently out of the question. The same applies to vehicles which are part of the coordinated transportation services provided by the SoS: roadworthiness of the novel properties and supporting sub-systems has to be demonstrated before a vehicle joins the operational SoS.

3.5 Test Case Generation in Presence of Dynamic Object Creation / Deletion

[GPGG08] point out that dynamic object creation and destruction is a crucial aspect of SoS behaviour – think, for example, of ships joining and leaving an SoS configuration of cooperating vessels. From testing and model checking object-oriented systems it is well known that this conceptually unbounded state space of dynamically generated objects represents an additional challenge: it has to be determined which numbers of objects considered in test configurations are meaningful to ensure that the SoS behaviour will be appropriate in any configuration that may occur.

[GPGG08] further point out that dynamic object creation implies that SoS should be prepared for mandatory run-time acceptance testing: any new object joining should be subject to run-time tests ensuring that the new component does not impair the services of the old configuration which are expected still to be functional in the extended configuration. Moreover, arrival of a new object may extend the capabilities of the resulting SoS, and these new capabilities have to be tried out before becoming operative.

These considerations induce further partitioning requirements regarding the effects of run-time tests which must not lead to a degradation of the SoS operational functions while the tests are carried out: all effects – whether faulty or intended – of these tests have to be segregated carefully from the state space and interfaces utilised by the operational SoS functions concurrently performed.

Observe that run-time testing is a common feature in vehicles, trains control systems and aircrafts; in the latter domain the term built-in test equipment (BITE) was coined for all components contributing to these run-time tests. The extent of BITE tests suggested by [GPGG08], however, reaches far beyond the typical BITE functions available, say, in today’s civil aircrafts: for SoS BITE tests have to dynamically consider the addition or reduction of capabilities, while conventional avionic BITE always focuses on monitoring a
pre-defined set of hardware and software-based functionality.

Due to the different stakeholders present in SoS operations it will frequently occur that data exchange between constituent systems may be subject to security restrictions. In these situations test data may represent covert channels allowing protected data to be passed between constituent systems, together with the test evidence passed between constituent systems in order to gain acceptance for cooperation within given SoS configurations: test cases, pass/fail results and test execution logs may directly carry classified information or allow the deduction of such information from the data.

### 3.6 Model-Based Testing

The guideline [Dep08] recommends a model-based approach to SoS development and V&V, since semantically well-defined models present a clearer view of system capabilities than informal descriptions, and form the basis for automated development and V&V activities. The benefits of model-based testing (MBT) have been clearly identified [BDG+08], and case studies show the feasibility of the approach for industrial-size systems [PHL+11]. A particular benefit of the MBT approach consists in the possibility to derive relevant test cases automatically from the model. Moreover, if model elements are already linked to requirements, as supported, for example, by the SysML modelling formalism [Sys10], MBT can trace requirements to test cases and procedures in an automated way. The feasibility proof, however, currently applies to sub-systems only: as of today, there exists no accepted methodology describing how to combine constituent system test models into an SoS testing model. A simple parallel combination of these test models would certainly lead to global SoS representations of intractable complexity, as indicated by the considerations in Section 3.1 above. Moreover, there exists no accepted methodology for deriving test cases for emergent SoS properties from such a model, even if its complexity could be handled.

Model-based testing for small- to medium-sized control systems may be fully automated, with potential additions of user-defined test cases that may be specified, for example, already as parts of the test model [PHL+11]. Typical success stories from this MBT application area emphasise the importance of the availability of a complete test model [LP10] as starting point of the test automation tool chain. This availability may turn out to be a major obstacle when it comes to model-based SoS testing: due to the number of cooperating constituent systems for which sub-models have to be contributed to the SoS
test model, and due to potentially under-specified SoS capabilities we expect that a complete SoS test model will only be available at the later stages of the development life cycle. As a consequence, the benefits of automation are in danger of being nullified by the disadvantage of delayed start of SoS system-level testing, if the tool chain depends on the availability of a complete test model. We conclude that for these reasons it is mandatory to support incremental development cycles for test models and test objectives, such that each cycle may be concluded with a test generation and execution campaign. Conversely, these cycles may be started by executing manually designed tests whose results can serve as inputs to the construction of the next model increment. The early cycles will use simulated equipment only, in order to avoid the costs of using original equipment and test facilities in stages where only few test cases are well-defined and many mission threads or scenarios are still in their development phase.

The need for incremental and even partially automated test model development has gained attention in the research communities, and [Vaa12] (see also the references given there) describes how machine learning techniques can be applied to automatically construct test models by incrementally deriving state machine models from test execution traces observed. This approach is obviously attractive for MBT, since testing already starts while the model is constructed, and it may be expected that model elaboration is accelerated by a combined strategy using automated machine learning and manual model design. According to the current state of the art of automated model construction, however, only single sequential (extended) state machines can be tackled, whereas the complexity of SoS will certainly require the construction of concurrent real-time models: it will be infeasible for most SoS to represent a sufficiently comprehensive test model as the sequential product automaton built from its concurrent state machines.

Summarising the challenges discussed above in the light of SoS characteristics, the distribution, the interdependencies and the interoperability required for typical SoS contribute to the complexity of SoS test strategies. Furthermore, there is no widely accepted method describing how to elaborate test cases focused on testing emergent behaviour. Finally, SoS size and the evolution of SoS capabilities over time suggest incremental and partially automated test model construction, which is currently only supported for sequential systems of medium complexity, while test model construction for SoS requires concurrent real-time approaches that are able to cope with high complexity.
Chapter 4

Towards a Strengthened Discipline of SoS Testing

4.1 Overview of Anticipated Research Fields

We expect that SoS testing will become a specialised discipline of model-based testing, since the MBT approach offers the best basis for test automation, as well as for justification of the test cases exercised on the SUT. SoS MBT will be supported by several research fields which are already well established or at least well identified today and serve for other testing areas apart from SoS testing as well: (1) techniques for systematic consolidation of different standards applicable for V&V in the different constituent system domains, (2) risk-based testing methods to reduce the unmanageable amount of interoperability tests that might be useful to perform, (3) simulation methods to replace original equipment in order to reduce costs and experiment with variants of behaviours that otherwise might require intrusive (and even destructive) test methods, and (4) investigations under which circumstances it may be acceptable to apply defect prediction techniques based on errors detected in the tests performed so far [HV10].

Other research areas, however, seem to explicitly emerge from the investigation of model-based SoS testing and its specific needs. Based on the test-related challenges identified in Chapter 3 we anticipate and advocate in-depth research on the following topics in the next years.

\[1\] Observe that at least for safety-critical constituent systems from the avionic, railway or automotive domains it is currently not accepted to base end-of-testing criteria on statistic defect prediction techniques.
1. SoS-specific Formalisms for model-based testing.
2. Identification of test objectives for emergent properties.
3. Methodology for incremental model-based SoS testing.

These research fields will be sketched in more detail in the sections below.

4.2 SoS-Specific Formalisms for Model-Based Testing

The objective of test models is to present expected SUT behaviour at a level of abstraction that is suitable for the purpose of testing. As a consequence, modelling details are only relevant as far as they describe the possible interactions and observations to be expected on the interface between test environment and SUT. The structural aspects of a test model need not have any relationship to the more fine-grained structural properties of development models elaborated for the SUT. Nevertheless formalisms for model-based SoS development and for MBT require certain similar capabilities to be addressed in current and future SoS-related research, in particular the need for abstraction and for confidentiality.

Due to SoS complexity, it will generally not be possible to specify emergent properties by reference to the detailed models of the constituent systems involved. This will be far too complicated and also in several cases not desirable, because some stakeholders of constituent systems may not be willing to disclose internal model details to collaborating SoS sub-system developers. As outlined in more detail by [CML+12], these considerations lead to the application of contract-based specification models: contracts specify the capabilities of constituent systems in such a way that (1) the emergent SoS properties can be clearly derived from the collection of constituent system contracts and (2) only the non-confidential capabilities are revealed. For test purposes we expect that the notion of contracts will be further extended to comprise information about V&V results already obtained on constituent system level: on the one hand, as pointed out in [LR09], certain portions of component-related tests should be re-executed during SoS testing in order to verify that the constituent system functionality is not compromised by co-operating systems in the SoS context. On the other hand it is argued in [MPS11, pp. 18] that the main focus of SoS system integration testing should be on the emergent properties. Therefore it is desirable to evaluate
knowledge about constituent system specific V&V results in order to cover complementary execution paths during SoS testing which have not been exercised before. The representation of V&V results already ascertained for the constituent system should be part of the test models’ contract descriptions. The diversity of V&V artefacts – from review results, formal analyses and test case specification to test execution logs – suggests a thorough investigation of suitable description techniques based on knowledge representation methods from artificial intelligence, in order to support automated analyses of these results for the purpose of identifying SoS test objectives.

Since the objective of contracts is to specify constituent system properties on a level of abstraction that is suitable for reasoning about SoS system-level capabilities, they may hide a considerable amount of details about the systems they describe. As a consequence contracts will frequently be nondeterministic due to under-specification.

While contracts are the suitable means to abstract constituent system capabilities, suitable formalisms for MBT on SoS level will also require language elements for specifying expected emergent behaviour, as far as not directly implied from the contracts. In particular, the rules for dynamic instantiation and deletion of constituent systems during SoS operation are specified on this level: in terms of object orientation, test models will only show classes and their relationships, but no object diagrams. The rules for dynamically creating and deleting constituent systems from these classes are part of the emergent property specification.

Finally, modelling formalisms for testing SoS need to allow for explicit definition of test cases by the test engineers, even for situations where full test automation support is available: test engineers need mechanisms to import their test-related domain expertise into the model, so that they can influence or complement the process of automated test case and test data generation. Test cases need to be traceable to contracts and from there to system-level SoS requirements [HHH+12].

According to our best knowledge, SysML [Sys10, HP08] is currently the only modelling formalism offering at least basic support for all of the items addressed above: requirements can be explicitly represented and traced to other model elements, contracts may be expressed as general or parametric constraints and test-related artefacts can be modelled as indicated in the UML Testing Profile UTP [BDG+08]. There exists, however, considerable potential for refining and extending the existing formalism with respect to expressiveness, semantic precision and standardised tool-oriented information representations. In particular, the Object Constraint Language OCL [CG12].
– this is the language subset for specifying datatypes, constraints and operations – in its present version does not support the specification of model computations, as for example, provided by temporal logic [MP81] or trace logic [Hoa78]. These are highly relevant for formally capturing test cases and requirements.

4.3 Identification of Test Objectives for Emergent Properties

The complexity considerations described in Section 3.1 indicate that test objectives for SoS system-level testing have to be specified on a higher level than that of explicit representations of interface data and state data as advocated in [LR09]: a higher level of abstraction is needed both for test objective specification and for test coverage assessment. From a practical point of view, these higher level objectives may be formulated as ordered collections of mission threads used for end-to-end-testing [CCT08]: mission threads (again, the term has been coined in the military domain) are chains of actions performed locally in constituent systems in order to provide some specified service. These threads may be combined in a partially ordered way to perform end-to-end functions of the SoS. In their weakest form – for example in presence of incomplete SoS requirements – objectives may be represented by collections of scenarios as described by Kaner in [Kan03]: these also refer to services performed by constituent systems, but their justification need not be based on formal requirements, but may rely on the testers’ intuition, the knowledge of domain experts, experiences with similar products, and so on.

From a more formal perspective, mission threads or scenarios on constituent system level identify collections of traces – that is, finite execution sequences of constituent system models – which are suitable to investigate the test objective under consideration. Each collection may be regarded as an equivalence class, originating from the more abstract concept of mission threads or scenarios, instead of the concept of input, output or structural classes. [GGSV02] introduce a fairly general concept of equivalence classes based on action systems which is suitable as a formal basis for the mission thread / scenario-based class approach.

Using these concepts, the combinatorial testing problem on SoS system level will no longer consist of finding suitable sequences of input vectors, but
on identifying ordered collections of mission threads serving to verify given emergent properties. Constituent system contracts in test models need to specify the mission threads available and how to create concrete test data to stimulate thread executions.

[CCT08] advocate to complement end-to-end testing by tests against so-called net-readiness objectives: this is a variant of conformance testing, with the main objective to verify whether constituent systems support the visibility, accessibility and further properties of the data to be provided on SoS system level. This approach attempts to strengthen the compositionality argument discussed in Section 2.1 by testing each constituent system with respect to the compatibility with the convergence protocol of the overall SoS.

4.4 Methodology for Incremental Model-based SoS Testing

In the field of test model development methodology we consider the problem of incremental model development outlined in Section 3.6 as one of the main challenges of model-based SoS testing. The current methods based on sequential state machines as described by [Vaa12] may be extended to partially automated approaches where test model designers provide – apart from interface descriptions – initial architectural frames and suggestions for internal state variables, and automated machine learning takes these information into account. Furthermore, the explicit state machine construction may be complemented by incremental elaboration of transition relations: as pointed out by [PVL11b] for the purpose of test data generation, concurrent real-time models with complex state space are often better expressed by means of their transition relation than by explicit concurrent state machines. Promising attempts to construct test models in an incremental way from actual observations obtained during SUT simulations or experiments with the actual SUT indicate that test model development can profit from “re-engineering” SUT properties or model fragments from observations [RKF+09].

Finally, the importance of user-defined test scenarios will be much higher than for medium size control systems, because under-specified partial test models will not allow for the derivation of every test objective with its expected results in a fully automated way. We expect that new paradigms for the interaction between testing experts and test automation tools will be investigated in the near future, taking into account the utilisation of new model increments,
and allowing users to delegate constraint solving and combinatorial tasks for concrete test data generation to the test automation systems, while taking control of the test generation process whenever expert domain knowledge should be used to refine test cases and expected results.

Summarising the observations above, SoS characteristics influence the research on SoS testing methodology in the following ways: the autonomy and independence of constituent systems can be exploited to perform highly automated “local” model-based test campaigns for each constituent system. The complexity problems resulting from SoS distribution, interdependency and interoperability can be mitigated by means of contract-based test modelling on SoS level. The contract concept also facilitates to specify emergent SoS behaviour in a concise way. In the light of overall complexity and evolution of SoS properties over time, incremental test modelling will become a major research focus for SoS test model development.
Part II

SoS Testing Methodology
Chapter 5

The Model-Based Testing Paradigm

5.1 Model-Based Testing

In the COMPASS project model-based testing support is provided by the RT-Tester tool with its MBT component RTT-MBT. RT-Tester is an industrial strength tool for model-based testing of reactive concurrent real-time systems [PVLZ11, Ver07].

Following the definition currently given in Wikipedia:

Model-based testing (MBT) is the application of Model based design for designing and optimally executing the necessary artefacts to perform software testing. Models can be used to represent the desired behaviour of the System Under Test (SUT), or to represent the desired testing strategies and testing environment.

In this definition only software testing is referenced, but it applies to hardware/software integration and system testing just as well. Observe that this definition does not require that certain aspects of testing – such as test case identification or test procedure creation – should be performed in an automated way: the MBT approach can also be applied manually, just as design support for testing environments, test cases and so on. This rather unrestricted view on MBT is consistent with the one expressed in [BDG+08].

Automated MBT has received much attention in recent years, both in

\[\text{http://en.wikipedia.org/wiki/Model-based_testing}, \text{(date: 2012-06-14).}\]
academia and in industry. This interest has been stimulated by the success of model-driven development in general, by the improved understanding of testing and formal verification as complementary activities, and by the availability of efficient tool support. Indeed, when compared to conventional testing approaches, MBT has proven to increase both quality and efficiency of test campaigns; we name [LP10] as one example where quantitative evaluation results have been given. In this report the term model-based testing is used in the following, most comprehensive, sense: the behaviour of the system under test (SUT) is specified by a model elaborated in the same style as a model serving for development purposes. Optionally, the SUT model can be paired with an environment model restricting the possible interactions of the environment with the SUT. A symbolic test case generator analyses the model and specifies symbolic test cases as logical formulas identifying model computations suitable for a certain test purpose. Constrained by the transition relations of SUT and environment model, a solver computes concrete model computations which are witnesses of the symbolic test cases. The inputs to the SUT obtained from these computations are used in the test execution to stimulate the SUT. The SUT behaviour observed during the test execution is compared against the expected SUT behaviour specified in the original model. Both stimulation sequences and test oracles, i.e., checkers of SUT behaviour, are automatically transformed into test procedures executing the concrete test cases in a model-in-the-loop, software-in-the-loop, or hardware-in-the-loop
Observe that this notion of MBT differs from “weaker” ones where MBT is just associated with some technique of graphical test case descriptions. According to the MBT paradigm described here, the focus of test engineers is shifted from test data elaboration and test procedure programming to modelling. The effort invested into specifying the SUT model results in a return of investment, because test procedures are generated automatically and debugging deviations of observed against expected behaviour is considerably facilitated because the observed test executions can be “replayed” against the model. Moreover, V&V processes and certification are facilitated because test cases can be automatically traced against the model which in turn reflects the complete set of system requirements.

In Fig. 5.1 the MBT paradigm is sketched as described in Wikipedia as referenced above. There the term abstract tests is used for symbolic test cases, and executable tests for test procedures.

### 5.2 Development Models Versus Test Models

The reference model used for a MBT campaign may

- coincide with the model used to generate the SUT code according to the model-driven development approach, or
• consist of a separate model developed by the V&V team, as depicted in Fig. 5.2. In the former case tests are derived from the development model, so there are no possibilities to uncover logical, functional errors during testing, because the tests just ensure that the SUT behaves in a way which is consistent with the development model. This MBT variant is useful if the development model has been exhaustively verified with respect to correctness and validated with respect to completeness. The objective of the test suite is then only to verify whether the generated code is a correct refinement of the development model.

The latter case is to be applied in situations where

• the development model cannot be trusted with respect to correctness and completeness, or

• the level of abstraction of the test model is unsuitable for the test objectives.

In automated model-driven development models often show a greater level of detail than needed for testing: development models need to capture internal task structures, communication channels and event handlers which are not relevant, for example, when designing a black-box testing campaign only monitoring or stimulating the hardware interfaces of the SUT. In these cases the V&V team creates separate test models from the development models and from their own interpretation of the requirements. As a consequence, the test model may be at a different level of abstraction than the development model and describe a SUT behaviour that deviates from the one captured in the development model. The test suite may detect both deviations between code and development model and logical, functional errors in the development model.

5.3 Model-Based SoS Testing

For MBT in the context of SoS it seems unlikely that test suites might ever be derived from development models in a direct way.

• The composition of all the development models of constituent systems would be far too complex to be evaluated as one global model.

• It will frequently be the case that the detailed development models of constituent systems will be unavailable for SoS testing, due to privacy
considerations.

As a consequence we consider the elaboration of separate test models to be the typical approach to model-based testing of SoS. The components of such a test model should consist of

- contract-based behavioural abstractions of constituent systems, where the level of abstraction to be selected is determined by the signals observable during SoS testing,

- information about V&V already performed locally on each constituent system, to be exploited in SoS test suite design.

The latter information is used in two complementary ways.

- To deliberately re-test situations already handled in constituent system tests, in order to verify that the SoS composition is free of unwanted side-effects.

- To design novel SoS testing scenarios where constituent systems are exercised in ways not covered already in the local verification activities.

5.4 Basic MBT Automation Techniques

The RT-Tester architecture supporting automated MBT is depicted in Fig. 5.3. Test models are parsed in textual format (typically XMI) by the parser front end, and the model is internally represented by an abstract syntax tree (AST), called the RT-Tester internal model representation (IMR). The test case generator traverses the AST and identifies relevant test cases based on the syntactic model representation (for more details about test case generation see Chapter 8). Test cases are represented in symbolic form, that is, as logical formulas \( G(s_0, s_1, \ldots, s_c) \) over consecutive state valuations \( s_0, s_1, \ldots, s_c \): any test objective can be encoded as a formula specifying the characteristics of a finite sequence of states (also called a trace) to be traversed for meeting this objective.

The transition relation generator traverses the AST with the objective to encode the model’s operational semantics as a transition relation \( \Phi(s_i, s_{i+1}) \) relating states \( s_i \) to their post-states \( s_{i+1} \). The internal encoding of the transition relation handles events as pairs of Boolean flags, so mixed traces of event occurrences and state changes can also be internally encoded as
Concrete test data is created by solving constraints of the type
\[ J(s_0) \land \bigwedge_{i=0}^{n} \Phi(s_i, s_{i+1}) \land G(s_0, \ldots, s_{n+1}) \]
using the integrated SMT solver SONOLAR \cite{PVL11a}. In such a formula, conjunct \( J(s_0) \) characterises the current model state from where the next test objective \( G(s_0, \ldots, s_{n+1}) \) should be covered. Conjunct \( \bigwedge_{i=0}^{n} \Phi(s_i, s_{i+1}) \) ensures that the solution of \( G(s_0, \ldots, s_{n+1}) \) results in a valid trace of the model, starting from \( s_0 \).

Finally the test procedure generator takes the solutions calculated by the SMT solver and turns them into stimulation sequences, that is, timed input traces to the SUT. Moreover, the test procedure generator creates test oracles from the model components describing the SUT behaviour.
Formulas of the type displayed above are called bounded model checking instances (BMC instances) in BMC models are verified in the vicinity of states $s_0$, whether an undesirable property $G(s_0, \ldots, s_{n+1})$ – for example, a safety violation – can be realised within $n$ transition steps from $s_0$. The difference between test generation and model checking lies in the expectation whether a solution for $G(s_0, \ldots, s_{n+1})$ can be found: for a reasonably defined symbolic test case many solutions of the formula should exist, while solutions of the formula during model checking always uncover a model error. Since the SMT solver is able to find solutions for BMC instances, we can exploit this both for test generation and for local verification of test models; the latter is explained in Chapter 13.

\footnote{The notion of BMC instances is inspired by the term SAT instance used for Boolean formulas whose solvability is to be checked by SAT solvers.}
Chapter 6

CML/SysML Test Models – Supported Language Subsets

6.1 CML and SysML

SysML [Sys10,HP08] is a powerful wide-spectrum modelling language supporting a wide field of application domains. In the SoS context of the COMPASS project, not all aspects of the language are supported for the purpose of model-based testing. This chapter describes the subsets of SysML and CML that are currently supported by the MBT tool (RTT-MBT) of the COMPASS tool suite, and gives a forecast of the SysML/CML set to be supported by the end of the project.

Recall from Section 5.4 that automated MBT requires to calculate the operational semantics of test models by generating their transition relations. The COMPASS modelling language CML [WCF+12] is a textual language for modelling SoS. It is sufficiently expressive semantically to explain the behaviour of SysML models by transforming them into CML [MCI+12] in a way that is consistent with the original SysML semantics specified in [Sys10]. As a consequence, MBT tools may alternatively operate on the original SysML semantics or on CML semantics only, when creating test suites from SysML models (Fig. 6.1). Given a SysML model $M_{\text{SysML}}$ we may either generate its transition relation directly, following the right-hand side semantic arrow. Alternatively, we can first translate $M_{\text{SysML}}$ into its syntactic CML equivalent $M_{\text{CML}}$ according to the translation rules given in [MCI+12], and calculate the operational semantics of $M_{\text{CML}}$ using the CML definition alone (left-hand side arrows in Fig. 6.1). We have chosen the first alternative for generating
the transition relation of SysML models. The SysML models, however, may utilise CML syntax for specification of operations and data items, as explained in the paragraphs below. Therefore the current version of the RTT-MBT SysML model parser accepts CML pre-/post-condition specifications in operations and CML datatypes in attribute declarations. The generic RTT-MBT architecture depicted in Fig 5.3 is accordingly specialised for SysML with CML operations and datatypes as shown in Fig. 6.2. The SysML model is expected in its serialised XMI representation (provided by Artisan Studio) and parsed with the help of the libxml2 library.

It is planned for a later stage of the project (year 3) to provide a separate RTT-MBT parser front end for reading and testing against textual CML specifications (Fig. 6.3). It is currently investigated whether it is possible to encode CML models in the RTT-MBT AST in such a way that the existing transition relation generator for SysML/CML can be re-used. Alternatively, a novel generator optimised for encoding CML operational semantics in transition relations will be developed.
Figure 6.2: Test procedure generation from SysML models with optional CML operation and data type specifications.
Figure 6.3: Test procedure generation from CML models.
6.2 CML Subset Currently Supported for Operations and Datatypes

6.2.1 Types

For the current version of the RTT-MBT tool, types are restricted to atomic CML types defined in Fig. 6.4. The planned support for lists in the SMT solver of RTT-MBT will enable support for more complex data types from the CML language in later versions of the tool. The basic concept for representing complex data structures in a way that can be handled by the SMT solver for the purpose of test data creation follows the concepts described in [SWD].

```plaintext
type declaration =
    'types', type definition, { type definition };

type definition =
    [ qualifier ], identifier, '=' type, [ invariant ];

qualifier = 'private'
    | 'protected'
    | 'public'
    | 'logical';

type = basic type
    | quote type

basic type =
    'bool' | 'nat' | 'nat1' | 'int' | 'rat' | 'real' | 'char';

quote type =
    quote literal;
```

Figure 6.4: Types of the CML expression language supported in the current RTT-MBT version

6.2.2 Expressions

Because the supported types are restricted to the subset defined in 6.2.1, the expressions are appropriately restricted to the subset depicted in Fig. 6.5.
expression = unary expression
  | binary expression
  | name
  | old name
  | symbolic literal;

unary expression = unary plus
  | unary minus
  | not

unary plus = '+';
unary minus = '-';
not = 'not';

binary expression =
  expression, binary operator, expression;

binary operator = arithmetic plus
  | arithmetic minus
  | arithmetic multiplication
  | arithmetic divide
  | arithmetic integer division
  | arithmetic mod
  | less than
  | less than or equal
  | greater than
  | greater than or equal
  | equal
  | not equal
  | or
  | and

arithmetic plus = '+';
arithmetic minus = '-';
arithmetic multiplication = '*';
arithmetic divide = '/';
arithmetic integer division = 'div';
arithmetic mod = 'mod';
less than = '<';
less than or equal = '<=';
greater than = '>';
greater than or equal = '>=';
equal = '=';
not equal = '<>';
or = 'or';
and = 'and';

Figure 6.5: CML Expressions supported in the current RTT-MBT version
6.2.3 Operations

In a SysML model, operations are not defined in a textual way. The operation name and its attributes like parameters and the return type are modeled and the CML syntax for function and operation declarations does not apply. The RTT-MBT tool supports all CML expressions defined in section 6.2.2 in operation bodies in a SysML model. Additionally pre and post conditions of an operation are supported and are included in the transition relation of a model. This allows the implicit definition of operations through post conditions.

The current tool version accepts operations as part of a class or a block. CML expressions can also occur in guard conditions and actions of transitions, as well as entry-, exit- or do-actions of state machine states.

6.2.4 Events

Both CML and SysML support the concept of events, but with different semantics.

- CML events are passed along channels, and their communication is unbuffered and synchronous, so that sender or receiver are blocked as long as the other communication partner is not ready to receive or provide the event.

- UML/SysML events can be sent and received asynchronously, following the trigger semantics specified in the UML [OMG11, 13.3.31].

The encoding of both UML/SysML events and CML events is already captured in the extension of the RTT-MBT transition relation generator developed for COMPASS (see Chapter 11).

6.3 SysML subset

In the model bases testing approach as supported by RTT-MBT, the test procedures are automatically generated from a testing model of the system under test (SUT). The behaviour of the environment can be described and restricted through a model of the test environment (TE). This section describes the subset of the SysML language that is supported by RTT-MBT.
6.3.1 Packages

Packages are used to structure the model. The test model for an Rtt-MBT test campaign can be part of a complex SoS model. Therefore, Rtt-MBT test models must always be encapsulated in a separate package that contains the SUT model as well as the TE model.

- The SYSTEM package contains the proper test model, structured into TE and SUT sub-models.
- The REQUIREMENTS package contains the collection of requirements applicable to the SUT. These requirements are traced to model elements to support automated generation of traceability data, as described in Chapter 10.
- The RT-Tester package contains a very lean profile with some types supporting the MBT approach implemented by RT-Tester.

6.3.2 System Components

UML classes and SysML blocks can be used to model system components in the test model. A hierarchy of system components is supported so that a test model can be structured into sub-components. The hierarchy of components is taken into account during name resolution to be able to distinguish between two sub-components with the same name that are located below different higher level components. The current Rtt-MBT version ignores all visibility attributes of the model objects. All system components are globally visible.

6.3.3 Variables

UML interfaces, UML class properties and SysML properties of SysML blocks can be used to represent variables in the test model. The same scoping rules for name resolution as for component names apply to variable names. The current Rtt-MBT version ignores all visibility attributes of the model objects. All variables are globally visible.
6.3.4 State Machines

The behaviour of the SUT and the TE in a test model is expressed in behavioural state machines. Each component or sub-component can own one or more state machines expressing the behaviour of this component. UML simple composite states (with only one region) and UML submachine states are supported to decompose the state machines. Composite states with more than one region (orthogonal states) are not supported by the current RTT-MBT version. For submachine states, only the default entry and exit points are supported.

6.3.5 Operations

Operations of UML classes or SysML blocks can be used to model functions and operations in the test model. The supported CML subset described in 6.2.3 can be used in the body of the operations. Operations can be used as guard conditions or actions of transitions or as entry, exit or do actions of states. Constraints can be added as pre or post conditions of an operation.

6.3.6 Data Types

The current tool version supports atomic CML types as described in 6.2.1. These types are modelled in the COMPASS-Profile that is available in the test model template. New data types can be defined in the test model. Note that the SMT solver of RTT-MBT currently does not support complex types. The definition of UML enumeration types is used to model CML enumerated types.

CML and UML both support constraints for data types. The current RTT-MBT version supports constraints that limit the range of data types during the definition of the test procedure generation context. Constraints added to data types in the SysML model are not supported, yet. These constraints for data types can vary from one test procedure generation context to another even if both are using the same test model. Later versions of the tool will support constraints of data types to limit the range so that a default constraint can be defined in the model which can be replaced by a specialized definition in the test procedure generation context definition.
6.3.7 Requirements

SysML requirements are supported by RTT-MBT, as well as satisfy relations to support automatic tracing of requirements. The test generation supports requirements linked through satisfy relations to states and transitions of behavioural state machines, as well as interfaces and value properties of blocks. Requirements can also be indirectly linked to system components (UML classes or SysML blocks). In this case, a constraint has to be created, tracing to the system component and linked to the requirement by means of the satisfy relation (see Chapter 10).

6.3.8 Constraints

Two types of constraints are currently supported by RTT-MBT

- LTL constraints that trace to a system component and satisfy a requirement, and
- CML constraints for pre and post conditions of operations.

As described in more detail in Chapter 10, the LTL constraints are used to add manually defined test cases to the ones that are automatically generated from the model. If an LTL constraint satisfies a requirement, the requirement is expected to be covered, if the traces produced by a test execution satisfy this LTL formula.

CML pre or post conditions of operations are taken into account during the generation of the transition relation generation of a model.

6.3.9 Signals and Events

UML signals, signal events and signal triggers are supported for transitions of behavioural state machines (Section 11.1).

6.3.10 Flow Ports and Item Flows

Flow ports and item flows can be used in SysML internal block diagrams to describe the usage of variables in a test model. This is helpful to the reader of the model and will be accepted by the parser, but will be ignored. It was planned to use flow ports instead of value parts of a block to model variables,
but flow ports do not allow the definition of default values. Default values are an important part of the test generation context definition and therefore the flow ports and item flows are no longer included in the XMI AST.
Chapter 7

Abstract Syntax Tree Representations for CML/SysML Test Models

The abstract syntax tree (AST) of a test model is the representation that all Rtt-Mbt tools take as input. It is a vital component in the workflow of model based testing with Rtt-Mbt. UML and SysML are standardised languages, but in practice the modelling tools for UML or SysML either implement different parts of the complete standards or vary slightly in the details how the standards can be interpreted. The AST that is generated by the Rtt-Mbt parser from a SysML model can be seen as a homogeneous model representation that is identical for models which have been developed with different tools, but are equivalent with respect to syntax and static semantics. Indeed, these different interpretations will most certainly occur, since the standards themselves admit so-called semantic variation points [OMG11, Sys10].

Observe that the assumptions about behavioural semantics made in different SysML/UML front-ends do not affect the test generation process in the Rtt-Mbt tool: test generation is based on the AST and a transition relation reflecting the operational semantics of the model (see Chapter 11). The transition relation is generated within Rtt-Mbt in compliance with the standards [OMG11, Sys10], but with its specific implementation decisions about semantic variation points.
Figure 7.1: Two strategies to generate an AST from a SysML model.
7.1 Generating the AST

Currently two different strategies are used to transform a SysML model into an AST. The two strategies are illustrated in Fig. 7.1.

One way is to generate a CML representation of the model and use a CML parser to build an AST from the CML model. This AST will be called CML AST in this section. This way is used by most of the COMPASS tools.

RTT-MBT already contains an XMI parser that can generate an AST from the XMI export of a model. This AST will be called XMI AST in this section. The RTT-MBT tools all work on the XMI AST. The RTT-MBT tools and the COMPASS CML tools can communicate through a server component of the RTT-MBT tool suite. Data exchange between the two AST representations is planned through this client/server link between the tools, but is not yet implemented.

7.2 Example

The following example illustrates how the generation of the XMI AST of a simple SysML model would look like. The model is an RTT-MBT test model of a simplified controller that realises left and right indication flashing as well as emergency flashing. A turn indication lever is used to stimulate left or right flashing, and an emergency flashing button is used to control emergency flashing.

Figure 7.2: The internal block diagram of SYSTEM
The block definition diagram in Fig. 7.2 only contains a block for the system itself. The structure of the system and its sub-components is modelled in a separate internal block definition diagram, to be able to model item flows between the components of the system.

The model is divided into the system components $SystemUnderTest$ (SUT) and $TestEnvironment$ (TE), represented in an internal block diagram below the root node $SYSTEM$. The former models the expected behaviour of the system under test. This component is used during the test generation to calculate stimulation, expected result and test coverage. The latter can be used to define the behaviour of the environment of the SUT during the test generation. If this component is empty, the test generator is not restricted in the stimulation to the SUT. Fig. 7.3 shows this composition of the test model.

The flow ports and item flows describe the stimulation and indication channels of the test setup. In addition to the flow ports that are defined for both system components, the root component contains value properties that are used to model global variables for stimulation and indications of the SUT. For example, the emergency flash button of the system is modelled as the Boolean block property $EmerFlash$ of the root component $SYSTEM$ and as two flow ports $EmerFlash$, one as part of each system component. Moreover, the $SYSTEM$ block may define SysML events to be exchanged between TE and SUT. Only the block property is represented in the XMI AST. The flow ports and the item flows are ignored.
The current RTT-MBT version requires the manual application of the stereotypes SUT2TE or TE2SUT to the block properties of SYSTEM to indicate whether the variable or the SysML event generated from this property is used for stimulation or indication. In future versions it could be possible to derive this information in an automated way, because the reader-writer analysis performed within RTT-MBT identifies inputs to the SUT by the fact that these events and properties ("variables") are only read within the SUT components, while SUT output properties and events are written to or generated, respectively, by behaviours of the SUT.

Fig. 7.4 shows the package view of the block structure of the system components in Artisan Studio.

The SUT component of the test model only contains a single sub-component FLASH_CTRL in this simplified model. Larger systems can be decomposed in multiple concurrent sub-components.

The internal block diagram of the block SystemUnderTest (Fig. 7.5) illustrates how requirements can be assigned to system components. To link functional requirements to system components, constraints have to be added that are linked to the requirements by means of the satisfy relationship and to the
Figure 7.5: The internal block diagram of SystemUnderTest

components by means of the trace-relationship (see Chapter 10 for further details). This is shown in Fig. 7.5

Behaviour is associated with leaf components (=blocks) which are associated with SysML state machines. Leaf components are interpreted to operate concurrently. The component FLASH_CTRL contains such a state machine defining its behaviour, as shown in Fig. 7.6. It contains an initial state, a simple state EMER_OFF and a submachine state EMER_ON.

If the EmerFlash stimulation variable becomes true (the emergency flash button is pressed), the system changes from state EMER_OFF to EMER_ON. If EmerFlash becomes false while the system is in EMER_OFF, it changes to EMER_OFF. The behaviour of EMER_OFF is defined in the operation doEmerOff that is an operation of the block FLASH_CTRL. This operation is used as the do action of the state. The behaviour of EMER_ON is defined in the substate machine EMER_ON and is displayed in Fig. 7.7.

The state machine EMER_ON contains an initial state and two simple states EMER_ACTIVE and TURN_IND_OVERRIDE. The first state defines the emergency flashing behaviour (both turn indicators are flashing). If the turn indication lever changes during emergency flashing, the system changes to the state TURN_IND_OVERRIDE, which defines the behaviour of only the stimulated flash light is flashing. In this example, the trigger event

\[3\] Recall that SysML state machines are equivalent to the behavioural state machines of UML.
Figure 7.6: The state machine FLASH_CTRL

*TurnIndLvrChanged* is used to model that the turn indication lever stimulation changes from its current value to a new value. The guard condition ensures that the value of the turn indication lever is not 0 (neutral).

The state machine diagram of *EMER_ON* also demonstrates how requirements can be linked to states and transitions. No constraints are needed in this case, because states or transitions can be directly associated with the behavioural requirements

- “finally this state is reached by the computation”,
- “finally this transition is taken in the computation”,

respectively (again, see Chapter 10 for further details).

The Rtt-Mbt parser generates an AST from the XMI export of a model. The hierarchy of system components is presented in Fig. 7.8. The figure shows only a part of the AST generated for this example up to the first state chart location object *FLASH_CTRL*. The rest of the AST is shown in Fig. 7.9 and 7.10.

An XMI AST always starts with an IMR\(^4\) root element. A valid XMI AST needs to have a root component that is referenced by the IMR element. This IMR root system component represents the *SYSTEM* block of the SysML model.

The global interface variables of the system are added as attributes of the root component (attribute[0] to attribute[3] in the graph). The events and signals that are defined on top level in the model (see Fig. 7.4) are also added to the root component (signal[0] and event[0]) and the system components *SystemUndertest* and *TestEnvironment* are added sub-components of the

\(^4\) *Internal model representation*, the name for the AST in Rtt-Mbt.
Figure 7.7: The substate machine EMER_ON

Figure 7.8: The AST for the system component hierarchy
root component (subCmps[0] and subCmps[1]). Signal definitions and event definitions within the package Model are added as attributes of the IMR root component AST object (signal[0] and event[0]).

All names are expanded to reflect the position of the variables in the component hierarchy and avoid name clashes.

The constraints tracing to the sub-component FLASH_CTRL are represented as separate AST child objects (constraint[0] and constraint[1]) of FLASH_CTRL. The identifiers of the requirements satisfied by these constraints are added as attributes to these AST objects.

The only state chart that is defined in this example test model is located in the SUT sub-component FLASH_CTRL. A root state chart location AST element is generated for each state chart in the AST. The root state chart location for the state machine FLASH_CTRL is part of the AST sub-graphs in Fig. 7.8 and 7.9. All states of the state machine are represented as state chart location objects in the AST and are child elements of the root state chart location (subLocations[0] to subLocations[2]). Each state chart location can have a number of emanating transitions leading to another state chart locations.

Transitions are represented as separate objects in the AST, together with
AST objects for guard conditions, trigger events and actions of the transition. The transitions in the state machine \textit{FLASH\_CTRL} only contain guard expressions and no trigger events or actions. These elements are depicted later in the substate machine \textit{EMER\_ON}.

The state chart location \textit{FLASH\_CTRL.EMER\_ON} connects the AST sub graphs from Fig. 7.9 and 7.10. In the state machine \textit{FLASH\_CTRL}, it is used as one of the state chart locations defining the behaviour of \textit{FLASH\_CTRL}, but it is also the root state chart location for the state machine \textit{EMER\_ON}.

The identifiers of the requirements that are satisfied by the states \textit{EMER\_ACTIVE} and \textit{TURN\_IND\_OVERRIDE} and the transition from \textit{TURN\_IND\_OVERRIDE} to \textit{EMER\_ACTIVE} are added as attributes of these AST objects.

Rtt-Mbt supports UML signal events (\textsc{OMG11}, section 13.3.25) as triggers on transitions in behavioural state machines. Fig. 7.11 defines the hierarchy of events defined in UML. Signal events represent the receipt of an asynchronous signal. In this case, the signal is an attribute of the event.
Transitions can use signal events as triggers. The state machine `EMER_ON` contains a trigger event at the transition from `EMER_ACTIVE` to `TURN_IND_OVERRIDE`. In the AST, this is represented by a signal event object (triggerEvent) that refers to a signal object (signal). Both transitions in `EMER_ON` contain guard expressions (`TurnIndLvr <> 0`, `TurnIndLvr = 0`) which are represented as expression tree objects in the AST (guard).

---

5Refer to Fig. 7.12 from [OMG11]
Chapter 8

Model-Based Test Case Identification

Test cases can be derived from the model in an automated way. The strategies needed for this purpose are described in this chapter. Moreover, test experts may define their own “special-purpose test cases”; this is explained in Chapter 9.

8.1 Computations, Traces and Model Coverage

In our context of model-based testing, the behavioural semantics of test models is expressed by the set of its computations, that is, its infinite sequences $c = s_0.s_1.s_2...$ of state changes that may be executed by the model in accordance with the rules of its operational semantics. Note that events and time are also encoded in the states $s_i$, so that each state is a complete snapshot of internal model state and interface activities (see Chapter 11).

A trace is a finite prefix of a valid model computation. When testing, only traces of the model can be stimulated and observed. An input trace is a finite sequence of inputs passed to the SUT model portion over its input interfaces. As described in Chapter 6, interfaces are realised by property values travelling over ports along item flows, and by signal events.

We say that a trace $\pi = s_0...s_n$ covers a structural model element (flow ports, blocks) if the state sequence triggers state changes or behaviours of the element under consideration. Examples for traces covering structure are...
- state sequences where the properties associated with a flow port change their value ("the trace covers the port"),
- state sequence leading to state machine transitions in a machine associated with a block ("the trace covers the block and its higher-level blocks").

Behaviour is expressed by operations and state machines in the model. Trace $\pi$ is said to cover a behaviour of the model if it triggers the operations and (potentially concurrent) state machine transitions realising this behaviour.

### 8.2 Model coverage strategies

Model-derived test strategies aim at covering certain parts of the model structure and (a subset of) its behaviours. In Chapter 10 we will show how model coverage can be related to requirements coverage. The basic model coverage strategies are well-defined and well-explored, see, for example [Wei10] for a comprehensive discussion. For concurrent real-time systems, however, and for large-scale SoS, test strategies still require active research. In Fig. 8.1, 8.2, 8.3 an overview of these strategies is given, and we will discuss them in the paragraphs below.

**BCS – Basic Control State Coverage.** This type of behavioural coverage aims at covering each basic control state of each state machine at least once. No additional objectives are made about concurrent control states or accompanying variable valuations when reaching the control state under consideration. Using the general test case formula pattern

$$J(s_0) \land \bigwedge_{i=0}^{n} \Phi(s_i, s_{i+1}) \land G(s_0, \ldots, s_{n+1})$$

introduced in Chapter 5, BCS leads to formulas of the type

$$J(s_0) \land \bigwedge_{i=0}^{n} \Phi(s_i, s_{i+1}) \land G(s_{n+1})$$

with

$$G(s_{n+1}) \equiv s_{n+1}(\ell)$$

where $\ell$ is the name of the BCS to be covered. Control states are interpreted as Boolean variables (see Chapter 11), so $s_{n+1}(\ell)$ evaluating to true means
that the corresponding state machine resides in location $\ell$ in the state reached after the $(n + 1)^{th}$ transition step. Using linear temporal logic \cite{CGP99} LTL, the goal can be expressed by

$$F \ell$$

Observe that the goal shall be reached in state $s_{n+1}$. This means that at least one transition step will be performed: if basic control state $\ell$ is already active in $s_0$, that is, if $G(s_0)$ holds, we do not need to solve a BMC instance, but can directly mark $G$ as covered. This observation applies throughout the paragraphs below where different types of goal will be introduced.

**TR – Transition Coverage.** Transition coverage aims at covering each transition $\tau$ of every state machine in the model. Again, no restrictions are made regarding variable valuations, control states and concurrent transitions to be performed when the one under consideration is triggered. The associated goal can be expressed as

$$G(s_{n+1}) \equiv s_{n+1}(\text{trigger}(\tau))$$

or in LTL as $F \text{trigger}(\tau)$

where $\text{trigger}(\tau)$ is defined in Chapter \ref{Chapter:Introduction} intuitively speaking, $\text{trigger}(\tau)$ specifies that the required event triggering $\tau$ is available in the event pool,
the guard condition evaluates to \texttt{true}, and no higher-priority transition is enabled in model state \( s_{n+1} \).

**MCDC – MC/DC Transition Coverage.** *Modified condition/decision (MC/DC) coverage* is a variant of transition coverage, where non-atomic guard conditions are evaluated in a systematic manner.

- If a transition guard has the structure \( a \land b \), then the transition should be tested several times, so that at least the condition valuations
  \[
  a \land b \\
  \neg a \land b \\
  a \land \neg b
  \]
  are covered.

- If a transition guard has the structure \( a \lor b \), then the transition should be tested several times, so that at least the condition valuations
  \[
  \neg a \land \neg b \\
  \neg a \land b \\
  a \land \neg b
  \]
  are covered.

For the general case, we express guard conditions \( g \) in conjunctive normal form of atomic propositions \( g^j_i \)

\[
g \equiv \bigwedge_{i=0}^{n} \bigvee_{j=0}^{k_i} g^j_i
\]
D34.1 - Test Automation Support (Public)

(this transformation is performed automatically by Rtt-Mbt) and define

- **Stability test goals.** A minimal number of terms \( \bigvee_{j=0}^{k_i} g_i^j \) – if possible for exactly one \( i \in 0, \ldots, u \) – evaluate to \text{false}, all others evaluate to \text{true}. The stability test is performed for as many \( i \) as possible.

- **Progress test goals.** All conjuncts \( \bigvee_{j=0}^{k_i} g_i^j \) evaluate to \text{true}, but in each conjunct only a minimal number of \( g_i^j \) – preferably just for one \( j \) – evaluate to \text{true}. The progress tests are performed for as many different combinations of \( g_i^j \) evaluating \text{true} as possible.

As test goals, stability tests can be expressed as

\[
G_i(s_{n+1}) \equiv s_{n+1} \left( e \land \neg \left( \bigvee_{j=0}^{k_i} g_i^j \right) \land \sum_{h=0}^{u} \left( \bigvee_{j=0}^{k_h} g_h^j \right) = u \right), \quad i = 0, \ldots, u
\]

or

\[
F \left( e \land \neg \left( \bigvee_{j=0}^{k_i} g_i^j \right) \land \sum_{h=0}^{u} \left( \bigvee_{j=0}^{k_h} g_h^j \right) = u \right)
\]

In these definitions \( e \) denotes the event triggering \( \tau \). Identifying Boolean values \text{true}, \text{false} with 1, 0, respectively, the term \( \sum_{h=0}^{u} \left( \bigvee_{j=0}^{k_h} g_h^j \right) \) evaluates to the number of conjuncts evaluating to \text{true} in state \( s_{n+1} \), so \( \sum_{h=0}^{u} \left( \bigvee_{j=0}^{k_h} g_h^j \right) = u \) specifies that all but one conjuncts evaluate to \text{true}. If there is no solution for this constraint, one can try to solve \( \sum_{h=0}^{u} \left( \bigvee_{j=0}^{k_h} g_h^j \right) \geq p \) for some suitable \( p < u \).

For progress tests, the goals look like

\[
G_{m_1 \ldots m_u}(s_{n+1}) \equiv s_{n+1} \left( \text{trigger}(\tau) \land \bigwedge_{i=0}^{u} \left( g_i^{m_i} \land \sum_{j=0}^{k_i} \left( \bigvee_{j=0}^{g_i^j} \right) = 1 \right) \right)
\]

or, in LTL,

\[
F \left( \text{trigger}(\tau) \land \bigwedge_{i=0}^{u} \left( g_i^{m_i} \land \sum_{j=0}^{k_i} \left( \bigvee_{j=0}^{g_i^j} \right) = 1 \right) \right)
\]

conjunct \( \text{trigger}(\tau) \) states that the transition will be triggered. Every conjunct of the guard condition has exactly one disjunct \( g_i^{m_i} \) which evaluates to \text{true}, all other ones evaluate to \text{false}; this is expressed by the constraint \( \sum_{j=0}^{k_i} g_i^j = 1 \). If possible, test cases should be generated for every combination of \( (m_1, \ldots, m_u) \in \{1, \ldots, k_1\} \times \ldots \times \{1, \ldots, k_u\} \). If no solution exists fulfilling \( \sum_{j=0}^{k_i} g_i^j = 1 \), then weaker constraints \( \sum_{j=0}^{k_i} g_i^j \leq c, c > 1 \) can be used.
**HITR – Hierarchic Transition Coverage.** For transitions $\tau$ emanating from higher-level control states, different underlying basic control states can be active when $\tau$ is triggered. Hierarchic transition coverage aims at exercising $\tau$ once for every underlying basic control state being active. With the notation introduced in Chapter 11, let $\tau = (\ell_0, p, e, g, \alpha, \ell_1)$ a transition of some state machine $sm$ and and

$$H = \{ \ell \in BCS(sm) \mid \ell \neq \text{start}(p_{sm}(\ell)) \land \ell_0 \in [\ell, sm] \}$$

Set $H$ contains all basic control states having $\ell_0$ as their ancestor and which are not start states (because the state machine never resides in a start state). The HITR requires to test all objectives

$$G_\ell(s_{n+1}) \equiv s_{n+1}(\text{trigger}(\tau) \land \ell), \; \ell \in H$$

expressed in LTL as

$$F(\text{trigger}(\tau) \land \ell), \; \ell \in H$$

**EQ – Equivalence Class Coverage.** The “classical” method of equivalence class partition testing is justified for applications where certain sets $A$ of states are processed in an equivalent manner, typically by using the same data transformation $f$ on each member of such a partition. In this situation, it is possible to select a few members from $A$ for testing whether an illegal mutation of $f$ has been implemented. As a consequence equivalence class partition testing has a relationship to mutation testing, where test data sets are expressly selected for the purpose of uncovering certain types of mutations.

For adequate test strength of equivalence class testing of reactive systems it is necessary to traverse on certain paths between classes. As a consequence, test objectives are of the form

$$G(s_0, \ldots, s_{n+1}) \equiv \bigwedge_{i=0}^{n+1} \left( s_i \left( \bigwedge_{p \in A_i} p \right) \right) \land \overline{G}(s_{n+1}, \ldots, s_{n+k})$$

where $A_i$ are equivalence classes, each characterised as a set of atomic propositions $p$ over the model variables. Test objectives of this type specify that the SUT performs a trace $\pi = s_0 \ldots s_{n+1}$, such that each $s_i$ is member of a given class $A_i$. After the last class has been reached in state $s_{n+1}$, additional condition $\overline{G}(s_{n+1}, \ldots, s_{n+k})$ specifies a trace continuation $s_{n+2} \ldots s_{n+k}$ that is suitable to uncover errors in the data transformation applied on members of $A_{n+1}$. Depending on the type of data transformations $f$, different numbers of
conditions $\overline{G}$ have to be applied, because a mutant $f'$ of data transformation $f$ may yield the same transformation results as $f$ for some states of $A_{n+1}$. The classes $A_i$ can be derived from the model AST in a syntactic way by collecting the constraints guarding behaviours using identical data transformations. The feasibility of traces $\pi = s_0 \ldots s_{n+1}$ visiting sequences $A_0 \ldots A_{n+1}$, however, has to be explored using an SMT solver.

In LTL such a test objective is expressed as

$$F\left( \bigwedge_{p \in A_0} p \land (X(\bigwedge_{p \in A_1} p) \land (\ldots \land (X((\bigwedge_{p \in A_{n+1}} p) \land \overline{G})\ldots)) \right)$$

A formalised approach to equivalence class partition testing has been worked out in \cite{GGSV02}, but without formal justification of the test strength of this method. Indeed, equivalence class partition testing can be applied to achieve exhaustive test results, so that any error in the SUT will be uncovered, if certain boundary conditions are met. This property can be applied to justify the partitions used in a testing campaign. This will be explored in detail in \cite{HPS14}.

For SoS testing the EQ strategy will be applied to identify equivalent behaviours of constituent systems (for example, classes of certain mission threads), so that only some representatives of each class have to be executed during SoS system integration tests.

**BV – Boundary Value Test Coverage.** The well-known technique of Boundary value testing is closely linked to EQ testing, since it specifies special candidates to be selected from each equivalence class partition: by taking states $s_i$ at the boundary of classes $A_i$, the error detection strength is generally increased, because typical bugs like the confusion of $<$ and $\leq$ can be uncovered using such boundary candidates.

For reactive real-time systems boundary value testing applies to timed traces $\pi = s_0 \ldots s_{n+1}$, so that as many states $s_i$ as possible should lie on the boundary of a class $A_i$. Boundary values come both from the value and from the time domain ("just before the timer elapses, the expected event is received ... ").

The logical formulas for expressing test objectives look like the ones defined for equivalence class testing objectives, but they additionally require that a
subset of visited states should lie at the boundary of their class.

\[
G_B(s_0, \ldots, s_{n+1}) \equiv \\
\wedge_{i=0}^{n+1} \left(s_i(\wedge_{p \in A_i}(p \land (i \in B \Rightarrow s_i(\text{boundary } A_i))))\right) \land \\
\overline{G}(s_{n+1}, \ldots, s_{n+k}), \quad B \subseteq \{0, \ldots, n+1\}
\]

Predicates boundary \(A_i\) can be mechanically generated from a syntactic analysis of the atomic proposition contained in \(A_i\).

For robustness testing the conditions are changed in a way that state \(s_{n+1}\) lies in a neighbouring class of \(A_{n+1}\) and there just at its boundary.

The BV strategy will also be explored in detail in [HPS14].

**MCDCHITR – MC/DC Hierarchic Transition Coverage.** This HITR variant exercises high-level transitions with different guard valuations, as defined for MCDC coverage above.

The model-based coverage criteria specified so far are well known and accepted in MBT, and some of them have close relationships with code coverage strategies. This is not surprising, since programs are models, too, so code coverage can be regarded as model coverage. The following strategy is still a research topic, and it has been specifically designed by the authors for testing large concurrent systems, where complexity does not allow the representation of the SUT model as a single state machine, built from the product of several concurrent state machines.

**BCSPAIRS – Basic Control State Pairs Coverage.** A model state \(s\) is a function mapping all symbols (variables, basic control states, time) from a set \(V\) to their current value in some domain \(D\). We also call \(s\) a *state vector* and optionally use vector notation \((s(v_1), \ldots, s(v_n))\) if \(V = \{v_1, \ldots, v_n\}\). In model state \(s\) the active *basic control state vector* is the vector \((\ell_1, \ldots, \ell_m)\) indexed of the number of concurrent state machines \(sm_i\) in the model, so that each basic control state of this vector is part of the active configuration in state \(s\), that is, if \(\wedge_{i=1,\ldots,m} s(\ell_i)\) holds.

Large concurrent systems consist of so many state machines that test suites could never cover all basic control state vectors (let alone all complete state

\footnote{If such a product construction were possible, test strategies introduced successfully for verifying sequential systems could be applied for the concurrent one, too.}
vectors) from the concurrent state machines involved. The basic control state pairs coverage considers pairs of state machines in writer/reader relationship. For these the possible combinations of control state pairs should be covered by test suites, since exercising all of these pairs is likely to uncover errors in the writer/reader interaction. Let \( RW \subseteq SM \times SM \) the relationship characterising writer/reader pairs. Then the BCSPAIRS strategy aims at covering all test cases of the form

\[
G_{(\ell_1,\ell_2)}(s_{n+1}) \equiv s_{n+1}(\ell_1 \land \ell_2), \quad (\ell_1, \ell_2) \in RW
\]

written in LTL as

\[
F(\ell_1 \land \ell_2), \quad (\ell_1, \ell_2) \in RW
\]

The BCPAIRS strategy is still ongoing research, more details including an analysis of its test strength and an utilisation of the orthogonal array method for covering the a subset of all basic control state vectors will be described in [HPS14].

**IC – Interface Coverage.** Interface coverage is achieved by changing the component values of a given interface. If the interface vector components are \((x_1, \ldots, x_n)\), the test goals are therefore

\[
G_J(s_n, s_{n+1}) \equiv \bigwedge_{i \in J} s_{n+1}(x_i) \neq s_n(x_i), \quad J \subseteq \{1, \ldots, n\}
\]

This goal specifies that for a given subset of interface components identified by indexes from \( J \) the valuation should change in the \((n+1)^{th}\) transition step. In LTL this is expressed as

\[
\forall i \in J : \exists d_i \in D : \mathbf{F} \left( \bigwedge_{i \in J} x_i = d_i \right) \land \mathbf{X} \left( \bigwedge_{i \in J} x_i \neq d_i \right)
\]
Input interface coverage should be achieved for as many subsets $J \subseteq \{1, \ldots, n\}$ as possible, since some combinations of simultaneous changes may provoke faulty SUT reactions. For SoS testing the dimension $n$ of interface vectors would generally be far too large to test all the $2^n - 1$ index combinations $J$. Here methods like pairwise testing in combination with orthogonal array are more adequate (see references given in Part I), but even the number of pairs will be too large for exhaustive coverage. We therefore apply data flow analysis techniques on the model in order to determine which pairs $(x_i, x_j)$ are jointly used in state machines or operations.

**BC – Block Coverage.** The lowest level of block coverage is achieved by triggering some behaviour associated with the block, such as a transition of an underlying state machine or the execution of an operation defined in the block. Since blocks and their descendants may be regarded as sub-models, all behavioural coverage criteria listed in this section may be applied locally to the block. This aspect is relevant in regression testing, where the test focus might be on single blocks of a model which has been changed since the last baseline. It is therefore adequate to allow MBT tool users to specify the strategies to be applied on a per-block basis (and not just for the whole model).

\[\text{2The use may either be directly, when referring to } x_i, x_j \text{ or indirectly when using some variable } v \text{ where previous assignments from } x_i, x_j \text{ have been performed.}\]
Chapter 9

User-Defined Test Cases and
Their Formal Representation

In Chapter 8 it was shown how relevant test cases can be derived from a test model’s AST in an automated way. In addition to this it is desirable to allow testing experts to define their own, so called user-defined test cases.

9.1 Test Case Specification Formalism

As described in Chapter 8, test cases specify traces \( \pi = s_0 \ldots s_n \) which are suitable witnesses to test a certain test objective. Obviously the BMC instance notation

\[
J(s_0) \land \bigwedge_{i=0}^n \Phi(s_i, s_{i+1}) \land G(s_0, \ldots, s_{n+1})
\]

introduced in Chapter 5, while being appropriate as input to an SMT solver, is not a suitable candidate for manual specification of symbolic test cases. In the previous chapter, however, we have seen that certain types of BMC instances may be equivalently expressed by LTL formulas.

While LTL formulas are well-suited to specify computations fulfilling a wide variety of constraints, it has to be noted that it is also capable of defining properties of computations that will never be tested in practice, because they can only be verified on infinite computations and not on finite trace prefixes thereof (e.g., fairness properties). It is therefore desirable to identify a subset of LTL formulas that are tailored to the testers’ need for specifying finite
traces with certain properties. This subset is called **SafetyLTL** and described in the next section.

### 9.2 SafetyLTL

SafetyLTL has been introduced in [Sis94], and is suitable for defining safety properties of computations, that is, properties that can always be falsified on a finite computation prefix. SafetyLTL can be syntactically characterised by the following rules.

- Negation is only allowed before atomic propositions (so-called *negation normal form*).
- Disjunction $\lor$ and conjunction $\land$ are always allowed.
- Next operators $X$, globally operators $G$ and weakly-until operators $W$ are allowed.
- Semantically equivalent formulas also belong to SafetyLTL.

Recall that the weakly-until operator is defined as

$$
\phi \ W \ \psi \equiv_{\text{def}} (\phi \ U \ \psi) \lor G \phi
$$

Using $W$, the more common until operator can be expressed by

$$
\phi \ U \ \psi \equiv (\phi \ W \ \psi) \land F \psi
$$

and for finite traces $\pi = s_0 \ldots s_n$ formula $F \psi$ can be expressed as

$$
\psi \lor X \psi \lor XX \psi \lor XXX \psi \lor \ldots \lor \underbrace{X \ldots X}_{n \text{ times}} \psi
$$

For finite traces $\pi$, fulfilling $G \psi$ means that $\pi$ does not violate this formula. We could use the notation

$$
\psi \land X \psi \land XX \psi \land XXX \psi \land \ldots \land \underbrace{X \ldots X}_{n \text{ times}} \psi
$$

as an alternative to $G \psi$. 

78


9.3 Encoding SafetyLTL Formulas as BMC Instances

Given a SafetyLTL formula representing a symbolic test case, we need to encode it as a BMC instance, in order to let the SMT solver create a witness of this formula which can be used as a concrete test case. From [BHJ+06] it is known how LTL formulas can be checked in a stepwise manner while unrolling the transition relation for a bounded number $k$ of steps. For this purpose, semantic operator $\langle \varphi \rangle^k_i$ states that formula $\varphi$ holds in state $s_i$ of a trace of length $k$. For the operators of the SafetyLTL, their semantics can then be specified inductively by

- $\langle G \varphi \rangle^k_0 = \bigwedge_{i=0}^k \langle \varphi \rangle^k_{i-1}$
- $\langle X \varphi \rangle^k_i = \langle \varphi \rangle^k_{i+1}$
- $\langle \varphi \ U \psi \rangle^k_i = \langle \psi \rangle^k_i \lor (\langle \varphi \rangle^k_{i-1} \land \langle \varphi \ U \psi \rangle^k_{i+1-1})$

where $k$ is the number of steps the transition relation is unrolled.

Using these rules a recursive algorithm can create a BMC instance from any given SafetyLTL formula.

**Example 1.** Consider SafetyLTL formula

$$\phi \equiv (x = 0) U (y > 0 \land X (G z = 1))$$

and suppose we are looking for a witness trace $\pi = s_0 \ldots s_n \ldots$ with a length of at least $n + 1$ or longer.

Then the SMT solver is activated with the following BMC instances to solve.

In step 0, solve

$$bmc_0 \equiv \left( \bigwedge_{i=0}^n \Phi(s_i, s_{i+1}) \right) \land s_0(y) > 0 \land \left( \bigwedge_{i=1}^{n+1} s_i(z) = 1 \right)$$

If this succeeds we are done: the solution of $bmc_0$ is a legal trace $\pi$ of the model, since $\Phi(s_i, s_{i+1})$ holds for each pair of consecutive states in $\pi$. Formula $\phi$ holds on $\pi$ because $y > 0$ is true in $s_0$ and $z = 1$ holds for states $s_1 \ldots s_{n+1}$, so the right-hand side operand of $U$ is fulfilled in the initial state of this trace.
Otherwise we try to get a witness for the following formula in step 1.

\[ bm_{c1} \equiv \left( \bigwedge_{i=0}^{n} \Phi(s_i, s_{i+1}) \right) \land s_0(x) = 0 \land s_1(y) > 0 \land \left( \bigwedge_{i=2}^{n+1} s_i(z) = 1 \right) \]

If no solution exists we continue with step 2.

\[ bm_{c2} \equiv \left( \bigwedge_{i=0}^{n} \Phi(s_i, s_{i+1}) \right) \land s_0(x) = 0 \land s_1(x) = 0 \land s_2(y) > 0 \land \left( \bigwedge_{i=3}^{n+1} s_i(z) = 1 \right) \]

and so on, until a solution is found or no solution of length \( n + 1 \) is feasible. \( \square \)
Chapter 10

Traceability – from Requirements to Test Execution Logs

Traceability is the property of a development that all of its artefacts can be related to each other, in order to justify their existence and to verify their correctness and completeness. Automated model-based testing allows for automated compilation of all test-related traceability data. This will be described in the present chapter.

10.1 Test-Related Artefacts and Their Traceability Relationships

In the context of conventional testing, the following artefacts have to be traced to each other. Traceability between artefacts is always bi-directional.

- Test cases shall be traced to requirements.
- Test procedures shall be traced to the test cases they execute.
- Test execution results shall be traced to test procedures and to test cases.

The existence of any test case is justified by at least one requirement which is (at least partially) verified by means of this test case. Recall that requirements can be high-level or low-level (using the wording of RTCA DO178C [WG-I]), so tests can be related to end-user requirements or to technical derived requirements (such as “the stack size suffices for the application tasks involved”)
needed to realise the high-level ones. The relationship between test cases and requirements is $n : m$, because one test case may contribute to the V&V of several requirements, while one requirement may need several test cases for its verification.

The test results need direct traceability links to both procedures and test cases: when a test procedure execution fails, this is caused by at least one test case implemented by the procedure having failed during the execution. Therefore the test execution logs also have to identify which of the test cases passed or failed, or could not be executed at all, due to malfunction of the SUT or the testing environment.

In the context of model-based testing, traceability has to be extended; but this extension can be exploited to automate the whole traceability data compilation.

- Model elements shall be traced to requirements.
- Model-based test cases shall be traced to model elements
- User-defined test cases shall be traced to requirements.

The existence of each model element is justified by its contribution to the realisation of a (functional, structural, or non-functional) requirement. As a consequence, specific model coverage contributes to testing the requirements related to the model elements covered. This will be analysed in more detail in the subsequent sections.

### 10.2 Tracing Model Elements to Simple Requirements

If a model correctly reflects a set of requirements, each requirement is realised by a (usually infinite) set of model computations. These computations in turn can be characterised by the sequence of model elements they cover. In the general case these coverage criteria can be quite complex, but they can always be specified by means of an LTL formula referring to sequences of state vectors including time. Practical evaluation of requirements specifications of embedded systems (a published example is given in [PHL+11]) has shown that the majority of requirements can simply be expressed by formulas

$$\bigvee_{i=0}^{k} F\phi_i$$
Pressing and releasing one of the emergency flash switches de-activates crash flashing

Unlocking the doors via remote control de-activates crash flashing

Figure 10.1: Tracing model elements to requirements.
where the $\phi_i$ are state formulas indicating

- coverage of a control state, or
- coverage of transition

Intuitively speaking, every computation finally fulfilling at least one $\phi_i$ is a witness of such a requirement. To cover the requirement completely, we need cases whose traces finally cover all $\phi_i$ in arbitrary order.

**Example 1.** Fig. [10.1] shows a state machine of the test model published in [PHL+11]. The machine specifies the logic of a car’s emergency flashing function in presence of a crash situation. As long as no crash situation occurs, the machine resides in control state CRASH_FLASHING_PASSIVE, and the emergency flashing function is controlled by the driver using the respective dashboard switch. As soon as a crash occurs the machine transits via IMPACT_PENDING to CRASH_FLASHING_ACTIVE, and the emergency flashing function is automatically activated. Now there are three options to de-activate emergency flashing again:

1. Pressing and releasing the dashboard switch (transitions $\rightarrow$ EM_SWITCH_PRESSED $\rightarrow$ CRASH_FLASHING_PASSIVE).
2. Pressing and releasing the redundant special purpose switch provided in special vehicles (transitions $\rightarrow$ EM_SWITCH_SPV_PRESSED $\rightarrow$ CRASH_FLASHING_PASSIVE).
3. Unlocking the doors via remote control (transition $\rightarrow$ CRASH_FLASHING_PASSIVE).

The first two options have been encapsulated in one requirement identified as REQ-CRASH-FLASHING-0003; the third option is associated with requirement REQ-CRASH-FLASHING-0004.

The computations witnessing REQ-CRASH-FLASHING-0003 can be specified in LTL as

$$\text{REQ-CRASH-FLASHING-0003} \equiv \ F(\text{EM}_\text{SWITCH}_\text{PRESSED} \land \neg \text{db}_\text{EMSwitch}) \lor \ F(\text{EM}_\text{SWITCH}_\text{SPV}_\text{PRESSED} \land \neg \text{db}_\text{EMSwitchSPV})$$
Requirement REQ-CRASH-FLASHING-0004 can be formalised by

\[
\text{REQ-CRASH-FLASHING-0004} \equiv \\
F(\text{CRASH_FLASHING_ACTIVE} \land \text{oc_CentralLockingRequest} \land \text{oc_CentralLockingStatus} == 1)
\]

All of these formulas specify that certain transitions shall be finally triggered. In SysML this fact is represented in a graphical way through the «satisfy»-relation drawn from the transitions to the respective requirement, as depicted in Fig. 10.1.
10.3 Tracing Test Cases to Simple Requirements

As described in Chapter 8, test cases covering control states and transitions with various additional conditions are automatically derived from the model. During this process, test cases are automatically linked to the model elements they cover (see Fig. 10.2 for an example). As a consequence, simple requirements traced back to collections of model element to be covered in any order can be automatically traced to these test cases. For the requirements analysed in Example 1, the resulting traceability matrix looks like the one displayed in Table 10.1.

Table 10.1: Traceability matrix for Example 1.

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Test By</th>
</tr>
</thead>
<tbody>
<tr>
<td>REQ-CRASH-FLASHING-0004</td>
<td>TEST-CASE-CRASH-FLASHING-0017</td>
</tr>
<tr>
<td></td>
<td>TEST-CASE-CRASH-FLASHING-0018</td>
</tr>
<tr>
<td>REQ-CRASH-FLASHING-0004</td>
<td>TEST-CASE-CRASH-FLASHING-0019</td>
</tr>
</tbody>
</table>

10.4 Tracing from the Model to Complex Requirements

Complex requirements are represented by disjunctions of LTL formulas that are not simply of the form “Finally model element is covered”. Instead they depend on covering different vectors of model elements and other state conditions simultaneously, and in a specific time-dependent order. Since the propositions involved may refer simultaneously to arbitrary model elements, a graphical visualisation by more complex variants of the «satisfy»-relation seems infeasible. As a consequence we specify the LTL formula

\[ REQ \equiv \bigvee_{i=0}^{k} F \phi_i \]

characterising the requirement in a SysML constraint. Since this constraint cannot be linked to a single model element, it is connected to the SysML block declaring (directly or through sub-ordinate blocks) all symbols referenced in
any of the $\phi_i$. As a connector the «trace»-association is used. The SysML constraint is then linked to the respective requirement by means of the «satisfy»-relationship.

10.5 Test Cases Tracing to Complex Requirements

For automated test case generation the SysML constraints containing LTL formulas characterising requirements are analysed and test cases $F\phi_i$ are created for all the disjuncts. The traceability data is compiled by following the «satisfy»-association from the constraint to the requirement.

10.6 User-Defined Test Cases

Even if requirements could be linked to single model elements it is allowed to specify additional, so-called user-defined test cases. Again they are represented as SafetyLTL formulas and allocated in SysML constraints as described above.
Chapter 11

Generating the Transition Relation

In Section 5.4 it has been explained how symbolic test cases are represented by bounded model checking instances of the form

\[ J(s_0) \land n \bigwedge_{i=0}^{n} \Phi(s_i, s_{i+1}) \land G(s_0, \ldots, s_{n+1}) \]

From chapters 8 and 9 we know how to create instances of \( G(s_0, \ldots, s_{n+1}) \) in an automated or manual way. It remains to show how the transition relation \( \Phi(s_i, s_{i+1}) \) can be generated automatically from given models. This is done in Section 11.1 for SysML models\(^1\) and in Section 11.2 for textual CML models.

The encoding of a transition relation specifying operational SysML semantics is fairly complex; therefore an informal description of our semantic interpretation is given in Section 11.1.1.

Throughout this chapter we assume that readers are familiar with the basic terms of UML/SysML as described in [OMG11, Sys10] and with the basic terms of CML [MCI+12].

\(^1\)Recall from Chapter 5 that we use separate generation algorithms for textual CML models and SysML models.
11.1 Transition Relation Generation for SysML Test Models

11.1.1 Informal Description of SysML Semantics

In this section an informal description of our semantic interpretation for the SysML subset supported by RTT-MBT is given. This interpretation is consistent with

- the SysML 1.2 specification [Sys10], and
- the UML 2.4.1 superstructure specification UML241.

The UML standard has to be referenced, since SysML delegates the semantic interpretation of behaviour state machines to UML [Sys10 p. 105].

Real-Time Semantics. The interpretation of real-time behaviour follows the widely adopted concept of discrete transitions and delay transitions, as, for example, used in the semantic interpretation of Timed CSP [Sch99] and Timed Automata [BK+08 p. 673].

- Time is dense; the model execution time \( \hat{t} \) is defined in \( \mathbb{R}^+_0 \), only one global time exists.
- Discrete transition are performed in zero time.
- Delay transitions are performed while a positive amount of time passes.
- Delay transitions may only occur when no discrete transition is possible any longer.
- Models admitting unbounded sequences of discrete transitions are illegal, because this corresponds to having an infinitely fast machine executing them in zero time. Such a situation is called a livelock.
- Moreover, models are illegal if they admit infinite computations performed in a finite amount of time, because that would correspond to a machine operating faster and faster. Such a computation is called a zeno path.

This behavioural concept is one possible semantic variation of SysML, though the formalism would also admit semantic variations where discrete transitions consume time, at least in the case of transitions involving communication [OMG11 p. 467].
Behaviours. Discrete and delay transitions may change the state space by means of behaviours. SysML introduces two types of behaviours [OMG11 p. 435]:

- *Executing behaviour* is performed by one object.
- *Emergent behaviour* results from the interaction of one or more participant objects.

In the SysML subset currently supported executing behaviour is specified in

- operations specified in blocks, and
- state machines associated with leaf blocks that are not further decomposed into lower-level internal block diagrams.

Emergent behaviour can be stimulated by changing property values and generating events used by other blocks to trigger executing behaviour.

Causes for Behaviour. In the SysML subset currently supported for test automation purposes behaviour of operations is always caused by state machine transitions or control states (do, entry, exit actions may invoke operations).

State machine behaviour (= transitions) is caused by

- signal event [OMG11 Fig. 13.12],
- change event [OMG11 p. 439], and
- relative time event [OMG11 p. 468]
- completion event [OMG11 p. 583]

occurrences, or by do activities executed in absence of these event occurrences (see explanation of do activity semantics below).

Signal events are atomic, interface data has to be communicated via attributes that are in item flows of a higher-level block contain both sender and receiver.

Change events are caused by guard condition valuations changing from `false` to `true`.

2In our approach, also interface events and interface data are captured in a global state space.
Relative time events ("after (340ms)") are per default related to the point in time when the control state $\ell$ from where an emanating state machine transition is triggered by the time event has been entered. If the relative time event refers to another control state $\ell'$ entered prior to visiting $\ell$, a time observation is associated with $\ell'$. Time observations can also be associated with transitions emanating from $\ell'$.

**Delay Transitions.** Delay transitions have to increase the model execution time by a positive amount, but never beyond the next point in time when a time event is generated, because this may trigger a discrete transition. Delay transitions keep the values of internal model variables and control states stable.

**Example 1.** To illustrate the semantics of change and completion events, consider the state machine fragment depicted in Fig. 11.1. Suppose that control state $s1$ is currently part of the active configuration. In absence of an event occurrence $e1$, or if guard $g1$ always evaluates to $\text{false}$, transition $s1 \xrightarrow{[g0]/\ldots} s2$ is triggered by a change event indicating that the valuation of $g0$ changed from $\text{false}$ to $\text{true}$.

When entering $s2$, its entry action is performed and do activities associated with higher-level states may be executed. Suppose that $g3$ was already $\text{true}$
before $s_2$ has been entered. Suppose further that the valuation of $g_3$ remains 
_true_ after completion of entry and do activities. Then there is no change 
event available for $g_3$. The completion of the entry and do activities, however, 
raises a _completion event_. If $g_3$ evaluates to _true_ at this point in time, 
transition $s_2 \rightarrow [g_3]$ can be taken. □

**Event Pools.** As we have seen above, all behaviours (even those for which 
no user-defined triggers exist) caused by state machines are triggered by 
events; it is said that these events have a _triggered effect_: raising such an 
event leads to its insertion into an _event pool_, and transitions are triggered by 
dispatching events from the pool [OMG11, pp. 449-450]. The UML standard 
regards the event pool data structure and its management as a semantic 
variation point. Moreover, when several non-conflicting transitions may be 
triggered by the same event (for example, transitions of concurrent state 
machines that may all be triggered by the same event), it is not determined by 
the standard whether all behaviours or just one of them is triggered [OMG11, 
p. 450].

For RTT-MBT we have determined these variation points in the following 
way.

- Each event pool is represented by a FIFO queue.
- When an event is dispatched from the pool, _all_ transitions of concurrent 
  state machines operating on this pool are triggered, if this event matches 
  their trigger conditions.
- If the next event dispatched from the FIFO queue realising the event 
  pool does not trigger any behaviour and is not deferred by the state 
  machine’s current active control state, it will be discarded [OMG11, 
p. 472].
- When a sequential behaviour sends a sequence of events into the same 
  event pool, the sending order is preserved in the pool.
- When two behaviours $b_0, b_1$ sending events to the same event pool 
  execute concurrently, the event sequences signalled by $b_0$ and $b_1$ may be 
  interleaved in the pool in an arbitrary way.

\[\text{Note that behaviours specified by operation invocations which may be part of a state} \]
\[\text{machine’s transition behaviour have immediate effect within the transition: there are no} \]
\[\text{special events causing the invocations.} \]
These decisions have the following consequences on test models and test oracles.

1. Multiple generation of events leads to multiple event occurrences with the same number as generated, so SUT’s counting event occurrences of a certain type can be tested.

2. It can be tested whether the SUT reacts correctly on the sequence of events generated by the same sequential sender object.

3. For event sequences generated by different concurrent behaviours, the SUT behaviour must not depend on the specific interleaving of these sequences in the pool, because the oracle may check against another order.

4. If the SUT executes to behaviours concurrently, the test oracles only check the sequence of the generated events on a per-behaviour basis, and not their interleaving.

Though being slightly less intuitive than consequences 1 and 2, the consequences 3 and 4 are also well-founded: concurrent behaviours are executed in zero time, so the interleaving of all actions involved is nondeterministic. If we expect deterministic reactions from a SUT, its behaviour must not depend on nondeterministic event sequences on the pool.

If one SUT state machine $s_0$ sends an event $e$ to another SUT state machine $s_1$, however, it is checked by the test oracles whether the events sent by $s_1$ as a consequence of $e$ triggering some behaviour of $s_1$ all occur after $e$ has been sent.

Events are declared in the scope of a block; a separate event pool is associated with each instance of such a block. At most one event is dispatched per pool. One pool is processed at a time, but consecutive discrete transitions may dispatch events from different pools in zero time.

The current version of RTT-MBT works with two pre-defined event pools.

- $q_0$ contains events exchanged between TE and SUT in both directions.
- $q_1$ contains events exchanged locally within the SUT.

**Transition Selection Algorithm for the SUT.** The UML standard [OMG11, p. 576] states that lower-level transitions of compound states have higher priority than higher-level ones that might be simultaneously
"By definition, a transition originating from a substate has higher priority than a conflicting transition originating from any of its containing states."

Furthermore, the standard defines the following rules for firing transitions [OMG11, p. 576].

"The set of transitions that will fire is a maximal set of transitions that satisfies the following conditions:

- All transitions in the set are enabled.
- There are no conflicting transitions within the set.
- There is no transition outside the set that has higher priority than a transition in the set (that is, enabled transitions with highest priorities are in the set while conflicting transitions with lower priorities are left out)."

Conflicting transitions are those which emanate from an active control state, are simultaneously enabled, and have the same priority. The resolution of conflicts is a semantic variation point.

The transition selection algorithm used by RTT-MBT to interpret state machines residing in the SUT portion of the test model is consistent with these rules, but refines them in the following way.

- Conflicting transitions are not allowed.

In the present version of RTT-MBT conflicts have to be resolved by adding conjuncts to guard conditions: since only one event from a given event pool is dispatched at a time, only transitions triggered by the same event can be in conflict. This means that either

- all conflicting transitions are triggered by a change event or a completion event (that is, they are not associated with a user-defined event), or
- all conflicting transitions are triggered by the same user-defined event.

In both situations the conflict can be resolved by addition guard conditions in the following way. Suppose that transitions

\[ s_2 \xrightarrow{e[0]}/... \rightarrow \ldots, s_2 \xrightarrow{e[n]}/... \]

4Observe that this interpretation is valid for UML/SysML, but differs from the original Statecharts interpretation defined in [HN96].
are potentially in conflict. Then the conflict can be resolved by changing the state machine to

\[ s_2 \xrightarrow{e | \neg g_0 \text{ and } \neg g_1 \text{ and } \ldots \text{ and } \neg g_n / \ldots}, s_2 \xrightarrow{e | g_1 \text{ and } \neg g_2 \text{ and } \ldots \text{ and } \neg g_n / \ldots}, \ldots, s_2 \xrightarrow{e | g_n / \ldots} \]

or any other suitable priority order expressed by these additional guard conjuncts.

In a future version of RTT-MBT this conflict resolution can be abbreviated by adding priorities as additional constraints to potentially conflicting transitions.

Observe that conflicting transitions are allowed for state machines acting as environment simulations in the TE block. They are resolved in a nondeterministic way by means of the SMT solver trying to reach a specified test objective, subject to the constraints imposed by these TE state machines.

**Deferred Events.** If the next event dispatched from the FIFO queue realising the event pool does not trigger any behaviour, but is deferred by the state machine’s current active control state \( (e/\text{defer} \text{ command}) \), it will not be dispatched and therefore not be discarded \([\text{OMG11}, \text{p. 563}]\). Instead, the next event in the pool (if any) will be dispatched. Deferred events keep their position in the event queue. They are dispatched from the queue as soon as either

- the event is at the head of the queue and it is no longer deferred in the current configuration, or
- it is at the head of the queue and can be trigger a transition.

**Run-to-Completion.** An event dispatched from a given event pool \( p \) has to be completely processed, deferred or discarded before the next event can be dispatched from the pool. For state machines this means that all behaviours associated with a transition must be performed, such as the exit and entry actions, as well as the behaviour associated with state machine transitions involved. A run to completion is performed in zero time and corresponds to exactly one discrete transition. As a consequence all operation calls have to terminate after a number of bounded steps, because otherwise this would result in a livelock.

Observe that several runs-to-completion (that is, several discrete transitions) may be performed consecutively in zero time, if the target configuration
reached by the previous run has a further event in the pool that can trigger the next run. As mentioned above, the model is illegal if an infinite number of consecutive runs-to-completion can occur.

**Example 2.** Consider again the state machine fragment from Fig. 11.1. Suppose again, that transition \(s_1 \rightarrow s_2\) is taken, as discussed in Example 1.

The run-to-completion associated with this transitions ends after the entry action (and potential do activities of higher-level states) have been processed, that is, when the completion event for entering \(s_2\) has been raised and entered into the associate event pool.

Let’s analyse the following situations (which are not the only ones possible, since additional events not considered here might reside in the event pool).

- Event \(e_2\) is the next to be dispatched from the event pool and \(g_2\) evaluates to true.
- The completion event is the next to be dispatched from the event pool and \(g_3\) evaluates to true.

In the former case, transition \(s_2 \rightarrow s_3\) is taken. In the latter case, \(s_2 \rightarrow s_3\) is taken. In any case, a new run-to-completion is performed. □

**Do Activities.** Do activities are performed while the control state executing the activity (“do / <activity>” command) is part of the active configuration. Do activities are automatically terminated as soon as the control state is left as a result of a state machine transition. Do activities can be associated both with basic control states and higher-level states; the do activities of higher-level states continue being performed if a lower-level transition is triggered.

For RTT-MBT only one restricted form of do activities is allowed; they are invoked by a “do / <operation-call>” command, and the operation body may only contain a finite sequence of assignments

\[
\begin{align*}
x_1 &= \text{expr1}; \\
x_2 &= \text{expr2}; \\
\ldots
\end{align*}
\]

where “\text{expr1, ...}” denotes an expression over variable symbols in the scope of the state machine. The interpretation of this assignment sequence is as follows:
It is continuously checked whether – due to changes in the expression valuations for \texttt{expr1}, ... – the right-hand side valuation of an assignment differs from the variable valuation of the left-hand side.

- If such a change occurs, the assignment sequence is executed again.

Execution of the body is (part of) a discrete transition. A delay transition may not occur as long as such a list of do-assignments still has to be executed. Therefore a do activity of this kind is illegal, if the right-hand side expressions would continuously change. They are only allowed to change at discrete points in time.

\textbf{Example 3.} The restricted form of do activities supported by RTT-MBT is illustrated in Fig. [11.2] The state machine fragment on the left-hand side of the diagram shows a do activity performed while control state \texttt{s0} is part of the active configuration. As explained above, the behaviour of the do activity is equivalent to a self loop as depicted in the right-hand side state machine fragment: whenever at least one of the left-hand side variable valuations differ from the associated right-hand side expressions, the assignments are performed again. Note that the sequence of assignments has the same interpretation as in conventional imperative programming languages like C, C++ or Java: the valuation of \texttt{x} in the right-hand side expression of assignment \texttt{y = 2*x}; is the one generated by the previous assignment to \texttt{x}. \hfill \Box
**Inputs TE → SUT.** Inputs from the environment to the SUT may only be placed at the end of a delay transition. This rule is motivated by the fact that, while processing an event \( e \) sent by the TE in zero time by a finite sequence of discrete transitions, novel events and input data changes must not influence the internal processing of \( e \). Moreover, the widely accepted concept of finite variability states, that SUT latency forbids to process inputs from the environment occurring in arbitrarily fast sequence.

It is possible for the TE, however, to

- change an arbitrary number of interface properties, and
- create a bounded number of events for each input pool of the SUT.

### 11.1.2 Formalised Abstract Syntax

For formally specifying the algorithms for transition relation generation a formal representation of the SysML AST introduced in Chapter 7 is needed.

When using SysML test models, structure is represented by block definition diagrams and internal block diagrams. Let \( C \) the set of these blocks. Model components (=blocks) \( c \in C \) are arranged in a hierarchic manner, so that a partial function \( p_C : C \not\rightarrow C \) mapping each component but the root \( c_r \) to its parent is defined. The domain of the function is \( \text{dom } p_C = C - \{c_r\} \). Root \( c_r \) is the internal block diagram representing SYSTEM as described in Chapter 7. Each component may declare variables and events, and hierarchic scope rules are applied in name resolution. Interfaces between test environment and system under test as well as global model variables are declared on the level of \( c_r \). When parsing the model, the scope rules are applied to all expressions, and unique variable symbol and event names are used from then on. Therefore we can assume for the remainder of this section that all variable names are unique and taken from a symbol set \( V \) with pairwise disjoint subsets \( I, E_I, O, E_O, M, E_M, E_Q, T \subset V \) denoting TE → SUT input variables \((I)\) and events \((E_I)\), SUT → TE output variables \((O)\) and events \((E_O)\), internal model variables \((M)\), internal model events \((E_M)\), event pools \(E_Q\) and timers \((T)\).

SysML events \( e \) are represented in \( E = E_I \cup E_O \cup E_M \) by unique integer values like members of an enumeration type. For each enumeration value \( e \in E \) a Boolean variable \( \delta_e \) is associated indicating whether the event is currently deferred or not.

The two event pools used for TE ↔ SUT communication and internal SUT communication, respectively, are denoted by \( \{q_0, q_1\} \subseteq E_Q \) The FIFO queues
qi realising the pool admit the following operations.

- \(i = \#q\) returns the length of the queue.
- \(e = \text{head}(q)\) is only defined for non-empty queues and returns the head without modifying the queue.
- \(e = \text{at}(q, i)\) is only defined for integers \(i \in \{0, \ldots, \#q - 1\}\) and returns the \(i^{th}\) element of the queue.
- \(q' = \text{enqueue}(q, e)\) appends event \(e\) at the end of the queue and returns the modified queue.
- \(q' = \text{dequeue}(q)\) removes the head of the queue and returns the modified queue.
- \(q' = \text{delete}(q, i)\) is only defined for integers \(i \in \{0, \ldots, \#q - 1\}\). It removes the \(i^{th}\) element and returns the modified queue.

Function\(\eta : E \rightarrow EQ\) maps each event \(e\) to its pool \(q = \eta(e)\).

For each sequential state machine \(s \in SM\) an auxiliary output queue is defined. These queues will be used in the construction of the transition relation for concurrent behaviours, in order to specify the effects of concurrent event pool insertions in a consistent way (see explanations below). Function\(\zeta : SM \not\rightarrow EQ\) maps state machines \(s\) to their output queues \(q = \zeta(s)\).

Function\(d_C : C \rightarrow \mathcal{P}(C); d_C(c) \mapsto \{c' \in C \mid p_C(c') = c\}\) defines the direct descendants of a component. Leaf components \(c\) satisfy \(d_C(c) = \emptyset\). Each leaf is associated with a state machine \(s \in SM\), where \(SM\) denotes the set of all state machines which are part of the model. Function\(sc : C \not\rightarrow SM; \text{dom } sc = \{c \in C \mid d_C(c) = \emptyset\}\) associates component leaves with state machines.

State machines \(s \in SM\) are composed of locations (also called control states) \(\ell \in L(s)\) and transitions\(\tau = (\ell, e, g, \alpha, \ell') \in \Sigma(s) \subseteq L(s) \times (E_I \cup E_M) \times G \times A \times L(s)\) connecting source and target locations \(\ell\) and \(\ell'\), respectively. Trigger event \(e \in E_I \cup E_M\) denotes a SysML event firing the transition if also guard condition
$g \in \text{Bexpr}(V)$ (Bexpr($V$) denotes the Boolean expressions over symbols from $V$) evaluates to $\text{true}$. Special symbols are used for

- $\varepsilon_0$ – the changed-event (see description above),
- $\varepsilon_1$ – the completion event.

The guard condition is a Boolean condition over symbols from $I \cup O \cup M \cup T$. It may be assumed that the guard condition is always defined, because it can be defined as $\text{true}$ if no guarding restrictions apply to the transition. For timer symbols $t \in T$ occurring in $g$ we only allow Boolean conditions $\hat{t} > t + c$ with constants $c > 0$. When resetting timer $t$, the current model execution time $\hat{t}$ is assigned to $t$. Therefore $\hat{t} > t + c$ evaluates to $\text{true}$ if at least $c$ time units have passed since $t$’s most recent reset.

Transition component $\alpha \in A = \mathbb{P}(V \times \text{Expr}(V))$ specifies the effect of the transition, which is a (possibly empty) behaviour ([OMG11] Fig. 15.2). We only consider simple terminating behaviours, so they may be formally represented by sequences of value assignments to variables in $V$, according to expressions from Expr($V$). We allow expressions involving operations in Expr($V$). These operations may (“functions”) or may not (“procedures”) return values. In the former case, assignments like $(v, z + f(v_1, \ldots, v_n))$ are allowed. In the latter case, the left-hand side of the assignment is empty, we can think of an auxiliary variable $\kappa$ storing the constant return value of a procedure call, so that the left-hand side is always defined, as in $(\kappa, o\text{p}(x_1, \ldots, x_n))$.

For a pair $a = (v, expr) \in \alpha$, $\text{var}(a) = \text{def} v$ and $\text{expr}(a) = \text{def} expr$ denote the projections on variable and expression, respectively. For events the assignments

- $(e, \text{send})$ – send event $e$,
- $(\delta_e, \text{true})$ – defer event $e$ (this typically occurs in the actions associated with control states – the $\text{e/defer}$ directive – and will not appear in the effect of a transition),
- $(\delta_e, \text{false})$ – do not defer event $e$ any longer (this corresponds to the absence of an $\text{e/defer}$ directive in a control state)

are defined.

For timer symbols $t \in T$ only resets $(t, \hat{t})$ are allowed. This is implemented by assigning the current model execution time $\hat{t}$ to the timer variable, so that all durations relative to this occurrence can be calculated as $\hat{t} - t$.

A transition without accompanying assignments is associated with an “empty
effect \( \alpha = \varepsilon \) (the empty sequence is denoted by \( \varepsilon \)).

Function

\[
\omega_s : L(s) \rightarrow \mathcal{P}(\Sigma(s)); \ell \mapsto \{(\bar{\ell}, e, g, \alpha, \ell') \in \Sigma(s) \mid \bar{\ell} = \ell\}
\]

maps locations to their outgoing transitions. Locations are associated with (possibly empty) do activities, entry and exit actions; these are captured by the mappings

\[
do_s : L(s) \rightarrow A, \quad \text{en}_s : L(s) \rightarrow A, \quad \text{ex}_s : L(s) \rightarrow A,
\]

For the top-level location \( s \) of state machine \( s \) we require \( \text{en}_s = \varepsilon, \text{ex}_s = \varepsilon \), but do activity on the level of \( s \) are allowed. Deferral of events and ending their deferral is encoded as part of the entry action as a sequence of assignments \( (\delta_{e_1}, c_1) \cdot (\delta_{e_2}, c_2) \ldots \) with \( c_i \in \{\text{true}, \text{false}\} \).

Control states can be decomposed hierarchically as OR-states\(^5\). The state machine \( s \) itself is identified with the top-level OR-state containing all other locations as subordinate control states. Function \( p_s : (L(s) - \{s\}) \rightarrow L(s) \) maps lower-level control states to their parent locations. \( p_s^n(\ell) \) denotes the \( n \)-fold application of \( p_s \) to \( \ell \), with \( p_s^0(\ell) = \ell \).

\[
d_s : L(s) \rightarrow \mathcal{P}(L(s)); \quad d_s(\ell) \mapsto \{\ell' \in L(s) - \{s\} \mid p_s(\ell') = \ell\}
\]

defines the direct descendants of a location \( \ell \). Leaf locations are called basic control states; \( \text{BCS}(s) = \{\ell \in L(s) \mid d_s(\ell) = \varnothing\} \) denotes the set of leaves belonging to state machine \( s \). To identify a hierarchy of locations, we introduce a recursive set definition

\[
\zeta_s(\ell, \ell') = \begin{cases} 
\ell & \text{if } \ell = \ell' \lor \ell = s \\
\varnothing \text{ else} & \zeta_s(p_s(\ell), \ell') \text{ endif}
\end{cases}
\]

and use notation \( [\ell..\ell'] = \zeta_s(\ell, \ell') \). If \( \ell' \) is removed from this set we write \( [\ell..\ell') \).

On each control state decomposition into sub-ordinate locations exactly one start location has to be identified. Start locations are basic control states with no incoming and exactly one outgoing transition which is unguarded.

Function

\[
\text{start} : (L(s) - \text{BCS}(s)) \rightarrow L(s)
\]

\(^5\)This term was used in Harel’s original Statecharts definition [HN96]. In the UML standard, the term non-orthogonal composite state is used [OMG11, p. 564].
maps higher-level control states to their direct descendants’ start locations. The target control state of the transition leaving a start location is called the initial location.

Given two locations \( \ell_1, \ell_2 \in L(s) \), the least common ancestor is the “closest” location containing both \( \ell_1 \) and \( \ell_2 \) as descendants. Function

\[
lca : L(s) \times L(s) \rightarrow L(s); \\
(\ell_1, \ell_2) \mapsto p^n_s(\ell_1) \quad \text{where} \\
n = \min\{m \in \mathbb{N}_0 \mid p^n_s(\ell_1) \in [\ell_1..s] \cap [\ell_2..s]\}
\]

defines this in a formal way which is consistent to the LCA definition given in [OMGI11, p. 574].

11.1.3 Symbolic Execution and Predicate Transformers

When deriving the complete transition relation for a SysML model, local sequential behaviours have to be captured as logical formulas describing the effect of a sequence of statements on the result state of discrete transitions. To this end, it is first explained in this section how the effect of simple assignment sequences may be specified by successive application of term replacements. This concept is usually called symbolic execution. In the case of operation invocations in these behaviours, conventional symbolic execution is no longer applicable, if these operations are specified by pre- and post-conditions which cannot be trivially transformed into assignments (like \( x' = x + y \)). For these situations symbolic execution is replaced by constructing strongest post-condition predicate transformers. Both techniques will be explained in this section.

Symbolic Execution by Term Replacement. For specifying the effect of behaviours in the transition relation the notion of term replacement has to be formalised, so that the operational semantics of sequences of assignments can be explained.

Let \( \text{Expr} \) the set of expressions over symbols from \( V \). A function \( r : V \rightarrow \text{Expr} \) is called a term replacement function. It maps every variable symbol or expression to the expression it should be replaced by. \( \text{id}_V : V \rightarrow \text{Expr} \) is

\( ^6 \)UML uses the term initial pseudo state for these locations [OMGI11, p. 594].
the identity function mapping each variable symbol to itself. Term replacement functions are extended to expressions \( e(z_1, \ldots, z_n) \in \text{Expr} \) by defining \( r(e(z_1, \ldots, z_n)) = e(r(z_1), \ldots, r(z_n)) \). For a behaviour \( \alpha = (z_1, e_1) \ldots (z_k, e_k) \) the resulting term replacement \( r_\alpha \) is defined recursively by setting

\[
\begin{align*}
    r_\alpha &= \rho(id_V, \alpha) \\
    \rho(r, \varepsilon) &= r \\
    \rho(r, (z, e) \cdot \alpha') &= \rho(r \oplus \{ z \mapsto r(e) \}, \alpha') \text{ if } z \in V - T \\
    \rho(r, (t, \text{reset}) \cdot \alpha') &= \rho(r \oplus \{ t \mapsto \hat{t} \}, \alpha') \text{ if } t \in T
\end{align*}
\]

In this definition \( \oplus \) denotes functional overriding: \( f \oplus g(x) = g(x) \) if \( x \) is in the domain of \( g \), otherwise \( f \oplus g(x) = f(x) \).

If a sequential behaviour is represented by a term replacement function \( r_\alpha \), then the strongest post-condition of this behaviour can be expressed as

\[ \forall v \in V : v' = r_\alpha(v) \quad (11.1) \]

so \( r_\alpha \) gives rise to a strongest post-condition predicate transformer. We call the terms \( r_\alpha(v), v \in V \) the result of the symbolic execution of \( \alpha \).

**Example 4.** Take, for example, a sequence of assignments

\[
\begin{align*}
t &= t_{\text{current}}; \\
x &= y + z; \\
y &= x + y; \\
z &= 2z + y;
\end{align*}
\]

representing some behaviour describing the combined effect of state machine transitions, entry actions etc. In our formal abstract syntax, this behaviour is expressed by

\[
\alpha = (t, \hat{t}).(x, y + z).(y, x + y).(z, 2z + y)
\]

The associated term replacement function \( r_\alpha \) is calculated by

\[
\begin{align*}
    r_\alpha &= \rho(id_V, \alpha) \\
    &= \rho(id_V, (t, \text{reset}).(x, y + z).(y, x + y).(z, 2z + y)) \\
    &= \rho(id_V \oplus \{ t \mapsto \hat{t} \}, (x, y + z).(y, x + y).(z, 2z + y)) \\
    &= \rho(id_V \oplus \{ t \mapsto \hat{t}, x \mapsto y + z \}, (y, x + y).(z, 2z + y)) \\
    &= \rho(id_V \oplus \{ t \mapsto \hat{t}, x \mapsto y + z, y \mapsto 2y + z \}, (z, 2z + y)) \\
    &= \rho(id_V \oplus \{ t \mapsto \hat{t}, x \mapsto y + z, y \mapsto 2y + z, z \mapsto 2y + 3z \}, \varepsilon) \\
    &= id_V \oplus \{ t \mapsto \hat{t}, x \mapsto y + z, y \mapsto 2y + z, z \mapsto 2y + 3z \}
\end{align*}
\]
so $r_\alpha$ replaces $t$ by $\hat{t}$, $x$ by $y + z$, $y$ by $2 \cdot y + z$, $z$ by $2 \cdot y + 3 \cdot z$, and leaves all other symbols unchanged. Expressed as a strongest post condition, this results in

$$t' = \hat{t} \land x' = x + z \land z' = 2 \cdot y + 3 \cdot z$$

Since concurrent behaviours may change the other variables from $V$, we do not assert anything for variables other than $t, x, y, z$. □

**Predicate Transformers.** As illustrated above, each term replacement function $r_\alpha$ gives rise to a predicate transformer specified in formula (11.1). If a behaviour involves operation calls and these operations are explicitly defined by means of “assignments in the operation body”, then this body can be copied into the assignments, initially assigning current values to the operation’s formal parameters.

If the operation, however, is only specified by means of pre- and post-conditions, it is only possible to specify the associated predicate transformer, while no symbolic execution term can be presented. To this end, we consider procedure and function invocations in behaviours $\alpha$, where it is not possible to replace operation calls by operation bodies, but instead strongest post condition predicate transformers have to be constructed.

For a behaviour $\alpha = (z_1, e_1) \ldots (z_k, e_k)$ containing such function calls, the associate strongest post-condition predicate transformer $sp_\alpha$ is defined recursively as shown in Fig. 11.3.

We assume that our symbol space $V$ also contains a collection of auxiliary variables and variable names of formal parameters occurring in operation declarations. We can further assume that all symbols are unique, so that no scope rules about different variables carrying the same name have to be observed. The predicate transformer is initialised in rule 1 with

- result predicate $\text{true}$,
- function $\xi$ assigning version 0 to every symbol, and
- the behaviour $\alpha$ to be processed.

The version identifiers $\xi(v)$ maintained for every symbol are needed to capture intermediate changes of a symbol $v$ in a versioned auxiliary symbol $v@\xi(v)$ while transforming behaviour $\alpha$ into a strongest post condition. The version numbers $\xi(v)$ are updated while processing $\alpha$.

In rule 2 the final processing step is described: if the behaviour has been completely processed, all variable symbols versioned with number 0 represent
1. \( sp_\alpha(\alpha) = sp(\text{true}, \{v \mapsto 0 \mid v \in V\}, \alpha) \)
2. \( sp(\phi, \xi, \varepsilon) = \phi[v \mapsto 0, v' \mapsto \xi(v) \mid v \in V] \)
3. \( sp(\phi, \xi, (z, expr).\alpha') = sp(\phi \land z(x(z)) = expr[v \mapsto \xi(v) \mid v \in V], \xi \oplus \{z \mapsto \xi(z) + 1\}, \alpha') \)
   for expressions not involving operation calls
4. \( sp(\phi, \xi, (\kappa, op_1(x_1, \ldots, x_n)).\alpha') = sp(\phi \land \psi, \xi', \alpha') \)
   where \( op_1() \) is declared as
   \[
   \begin{align*}
   & op_1(a_1 : t_1, \ldots, a_n : t_n) \\
   & \text{frame wr } w_1, \ldots, w_k \\
   & \text{pre expr}_0(a_1, \ldots, a_n, v_1, \ldots, v_m, w_1, \ldots, w_k) \\
   & \text{post expr}_1(a_1, \ldots, a_n, v_1, \ldots, v_m, w_1, \ldots, w_k)
   \end{align*}
   \]
   and
   \[
   \begin{align*}
   & \psi \equiv \text{expr}_0[x_1 \mapsto \xi(x_1)/a_1, \ldots, w_1 \mapsto \xi(w_1)/w_1, \ldots, v_1 \mapsto \xi(v_1)/v_1, \ldots] \land \\
   & \text{expr}_1[x_1 \mapsto \xi(x_1)/a_1, \ldots, w_1 \mapsto (\xi(w_1) + 1)/w_1, \ldots, v_1 \mapsto \xi(v_1)/v_1, \ldots]
   \end{align*}
   \]
   and
   \[
   \xi' = \xi \oplus \{w_1 \mapsto \xi(w_1) + 1, \ldots, w_k \mapsto \xi(w_k) + 1\}
   \]
5. \( sp(\phi, \xi, (\kappa, op_2(x_1, \ldots, x_n)).\alpha') = sp(\phi \land \psi, \xi', \alpha') \)
   where \( op_2() \) is declared as
   \[
   \begin{align*}
   & op_2(a_1 : t_1, \ldots, a_n : t_n) a : t \\
   & \text{frame wr } w_1, \ldots, w_k \\
   & \text{pre expr}_2(a_1, \ldots, a_n, v_1, \ldots, v_m, w_1, \ldots, w_k) \\
   & \text{post expr}_3(a_1, \ldots, a_n, v_1, \ldots, v_m, w_1, \ldots, w_k)
   \end{align*}
   \]
   and
   \[
   \begin{align*}
   & \psi \equiv (z \mapsto (\xi(z) + 1) = a \mapsto (\xi(a) + 1)) \land \\
   & \text{expr}_2[x_1 \mapsto \xi(x_1)/a_1, \ldots, w_1 \mapsto \xi(w_1)/w_1, \ldots, v_1 \mapsto \xi(v_1)/v_1, \ldots] \land \\
   & \text{expr}_3[x_1 \mapsto \xi(x_1)/a_1, \ldots, w_1 \mapsto (\xi(w_1) + 1)/w_1, \ldots, v_1 \mapsto \xi(v_1)/v_1, \ldots]
   \end{align*}
   \]
   and
   \[
   \xi' = \xi \oplus \{z \mapsto \xi(z) + 1, a \mapsto \xi(a) + 1, w_1 \mapsto \xi(w_1) + 1, \ldots, w_k \mapsto \xi(w_k) + 1\}
   \]

Figure 11.3: Strongest post-condition predicate transformer for sequential behaviours.
the pre-state, so we use unprimed symbol \( v \) wherever \( v@0 \) occurs. All symbols \( v \) have final variable version \( \xi(v) \), so all symbols \( v@\xi(v) \) are replaced by the post-state notation \( v' \). All remaining versioned variable symbols are auxiliary variables to be processed by the solver when calculating the effect of the behaviour.

In rule 3 an ordinary assignment not involving operation calls is processed. The existing postcondition predicate \( \phi \) is extended by a conjunct specifying the effect of the assignment. To this end, the left-hand side variable \( z \) of the assignment gets the incremented version \( \xi(z) + 1 \), because a new value is assigned to \( z \). The right-hand side expression is evaluated for all variables \( v \) occurring there with their most recent version number, that is \( v@\xi(v) \). For the next processing step, the versioning function \( \xi \) is updated for argument \( z \) which is now mapped to the new version \( \xi(z) + 1 \).

Rule 4 specifies how a procedure call of an operation \( op_1() \) specified only by pre- and post-conditions is processed. The current predicate \( \phi \) is extended by a conjunct \( \psi \) stating that the pre-condition \( expr_0 \) must hold when replacing all formal parameters \( a_i \) of \( op_1() \) by their actual parameters \( x_i \) in their actual version \( \xi(x_i) \). Global variables \( w_i, v_i \) possibly addressed in the pre-condition are also evaluated in their current version. The effect of the procedure call is specified by post-condition \( expr_1 \). Here the variable symbols \( w_i \) written to by \( op_1() \) are referenced with incremented version \( w_i@\xi(w_i) + 1 \). All other variables remain unchanged by the procedure execution, so they appear again with the most recent version. The versioning function \( \xi \) is updated for all the write-variables \( w_i \) which get incremented version numbers.

Rule 5 specifies the effect of an assignment involving a call to an operation \( op_2() \) returning a value to be assigned to the left-hand side variable of the statement. Here both the global variables \( w_i \) written to by the operation and the left-hand side variable of the assignment get new versions. The return value of \( op_2() \) is specified by an auxiliary variable \( a \).

**Example 5.** Consider a sequence \( \alpha \) of assignments

\[
\begin{align*}
y &= x + y; \\
z &= 2z + y; \\
z &= op(x,z) + y; \\
w &= z + w;
\end{align*}
\]

where the operation has been implicitly specified in CML style by

\[
\begin{align*}
\text{op}(a : \text{float}, b : \text{float}) & \ c : \text{float} \\
\text{frame wr w}
\end{align*}
\]
post \( w - a = b + w' \) and \( c = a*b; \)

Application of \( sp_{\alpha}(\alpha) \) results in the strongest post condition

\[
y' = x + y \land z_1 = 2 * z + y' \land \\
z' = c' + y' \land w_1 - x = z_1 + w \land c' = x * z_1 \land w' = z' + w_1
\]

Observe that the number of auxiliary variables used in predicate transformers can be reduced by applying term replacement functions “locally” on statement sub-sequences consisting of simple assignments without calls to operations specified by pre-/post-conditions only.

### 11.1.4 Transition Relation for SysML Models

**Overall Structure of the Transition Relation.** Using the abstract syntax introduced above, consider a model \( M = (C, SM, p_C, sc) \) with components \( c \in C \) in hierarchic order specified by \( p_C \) whose leaves are associated with state machines \( s \in SM = \{s_1, \ldots, s_n\} \) as specified by function \( sc \). The state of a model execution is specified by

- the **active basic configuration**, that is, the vector \((\ell_1, \ldots, \ell_n), \ell_i \in BCS(s_i)\) of basic locations where the state machines which are part of the model currently reside in,

- the current valuation of all variables from \( V \) (recall that events \( e \) are also represented in event pool variables \( q \)), and

- the current time \( \hat{t} \) of the execution.

We consider basic control states as Boolean variables \( \ell : \mathbb{B} \), with exactly one location per state machine \( s \) evaluating to \( \text{true} \), indicating that \( s \) currently resides in this location. Following [CGP99] we describe transition relations relating pre- and post-states by means of first order predicates over unprimed and primed symbols from \( BCS \cup V \cup \{\hat{t}\} \), where \( BCS = \text{def} \bigcup_{s \in SM} BCS(s) \). The unprimed symbols refer to the symbol value in the pre-state, and the primed symbols to post-state values.

The predicate \( \Phi \) representing a model’s transition relation can be structured as follows:

\[
\Phi(\sigma, \sigma') \equiv \text{def} \ ( (\text{trigger}_D(\sigma) \land \Phi_D(\sigma, \sigma')) \lor (\neg \text{trigger}_D(\sigma) \land \Phi_T(\sigma, \sigma'))) \land \text{Inv}(\sigma')
\]
\( \Phi(\sigma, \sigma') \) relates model pre-states \( \sigma \) to post-states \( \sigma' \), and as introduced informally above, a model transition is either a

- discrete transition \( \Phi_D(\sigma, \sigma') \) performed in zero time, or a
- delay transition \( \Phi_T(\sigma, \sigma') \) where a positive amount of time passes.

Predicate \( \text{trigger}_D(\sigma) \) specifies the condition that at least one discrete transition is enabled in the current model state. Delay transitions may only occur when no discrete transition is possible any longer. Therefore \( \neg \text{trigger}_D(\sigma) \) must be fulfilled for a delay transition to occur. At the end of each transition, some sanity conditions shall hold; these are expressed by predicate \( \text{Inv}(\sigma') \).

The initialisation condition for the model state ensures that \( \text{Inv}(\sigma) \) holds in every initial state.

**State Invariant.** Invariant

\[
\text{Inv} \equiv \text{def} \left( \forall s \in SM, \ell_0, \ell_1 \in BCS(s) : \ell_0 \land \ell_1 \Rightarrow \ell_0 = \ell_1 \right) \land \\
\left( \forall v \in V : v \in D_v \right) \land \\
\left( \forall q \in EQ : \#q \leq k \right) \land \\
\left( \forall q \in EQ, i \in \{0, \ldots, \#q - 1\} : at(q, i) \in E \right)
\]

expresses that each state machine must be in exactly one basic control state during each step of a model execution. Moreover, all variables \( v \) assume values in their domain \( D_v \) which is usually expressed as a range of values from a given super-type. Queues used as event pools must have bounded length\( ^7 \) less or equal to \( k \), and every element in the queue must have a valid event identification from \( E = E_I \cup E_M \cup E_O \). This invariant will be added to all transition specifications below.

**Initial State.** Each model execution starts in a state where

- all inputs variables to SUT have arbitrary values within their defined range,
- input events to the SUT may or may not be present in the event pool,
- all other variables are initialised by their default values, and

\(^7\)Since test data generation and model checking is performed by unrolling the transition relation only for a bounded number of times, the boundedness of event queues is an acceptable restriction.
• all state machines \( s \) are in their top-most start locations whose parent is the state machines \( s \) itself.

• All event pools have a completion event \( \varepsilon_1 \) as first element.

Formally,

\[
\text{Init} \equiv_{\text{def}} \text{Inv} \land \\
(\forall v \in V - \{I \cup q_0\} : v = \text{default}(v)) \land \\
(\forall s \in SM : \text{start}(s)) \land (\forall q \in E_Q : \text{hd}(q) = \varepsilon_1)
\]

By default, none of the events are deferred.

**Effect of Sequential Behaviours.** Given a behaviour \( \alpha = a_1 \ldots a_k \in A \) specifying the combined effects of state machine transitions, entry and exit actions, and do activities, all including operation invocations, its write set is defined by

\[
W(\alpha) = \{\text{var}(a_1), \ldots, \text{var}(a_k)\}
\]

The auxiliary variables \( \kappa \) associated as left-hand-side variables of procedure invocations are not inserted into write sets.

If \( \alpha \) does not involve invocations of operations that are specified by pre-/post-conditions only, the effect of \( \alpha \) is defined as follows, using the term replacement function \( r_\alpha \) introduced above.

\[
\epsilon(\alpha) \equiv_{\text{def}} \bigwedge_{z \in W(\alpha)} z' = r_\alpha(z)
\]

Observe that \( \epsilon(\alpha) \) does not specify that variables outside \( W(\alpha) \) remain unchanged, because behaviours executed concurrently to \( \epsilon(\alpha) \) might change those variables. The effect of sending an event is appending it to the component’s output pool. Let \( c \) the component identifier. Then

\[
\epsilon(e, \text{send}) \equiv (\zeta(c)' = \text{enqueue}(\zeta(c), e))
\]

The effect \( \epsilon(\varepsilon) \) of empty behaviours is defined to be \text{true}.

**Example 6.** For the behaviour \( \alpha \) specified in Example 4, the effect is calculated as

\[
\epsilon(\alpha) \equiv (t' = \hat{t} \land x' = x + y \land y' = 2 \ast y + z \land z' = 2 \ast y + 3 \ast z)
\]
If $\alpha$ contains invocations of operations specified by pre-/post-conditions only, its effect is described using the associated predicate transformer as described in Section 11.1.3.

$$\epsilon(\alpha) \equiv \text{def} \ sp_\alpha(\alpha)$$

All primed variables are inserted into the write set, but the auxiliary variables introduced is $sp_\alpha$ are not inserted into the write set $W(\alpha)$. Consider, for example, the predicate transformer application from Example 5. The resulting write set of the behaviour $\alpha$ analysed there is $W(\alpha) = \{w, y, z\}$.

**Trigger Conditions for State Machine Transitions.** Given state machine $s$, a transition $\tau = (\ell_0, p, e, g, \alpha, \ell_1) \in \Sigma(s)$ will be triggered if

- its source location or one of its subordinate control states is part of the basic configuration currently active,
- its guard condition evaluates to true in the current model state,
- the next event to be dispatched from the pool is $e$ and
- no lower-level transition of the location under consideration is enabled.

We denote the next event to be processed from the pool $d$. Then the trigger condition is captured formally by

$$\text{trigger}_s(d, \ell_0, e, g, \alpha, \ell_1) \equiv \text{def}$$

$$\exists \ell_0 \in BCS(s) :$$

$$\langle \ell_0 \land \ell_0 \in [\ell_0..s) \land (e = d) \land g \land$$

$$\langle \forall \ell \in [\ell_0..\ell_0) : \forall(\ell, \overline{e}, \overline{g}, \overline{\alpha}, \overline{\ell}_1) \in \omega_\ell(\overline{\ell}) : d = e \lor -\overline{g})$$

**Effect of State Machine Transitions.** Suppose transition

$$\tau = (\ell_0, e, g, \alpha, \ell_1)$$

For transitions which are not triggered by user-defined events (so they only have a guard condition) completion or change events $\epsilon_1, \epsilon_0$ are used as default triggers, so we can assume that $e$ is always defined.

Recall that we do not allow non-determinism for transitions emanating from the same control state. Therefore only transitions emanating from lower-level states may have higher priority than the transition under consideration.
between state machine locations $\ell_0, \ell_1$ will be triggered. Assume further that $\ell_0$ is the active basic control state equal to or subordinate to $\ell_0$. Formally,

$$\ell_0 \in BCS(s) \land \ell_0 \land \ell_0 \in \ell_0..s$$

Then the effect of state machine transition $\tau$ is defined by

$$\epsilon(\tau) = \epsilon(\alpha(\tau)) \land \iota(\ell_1)'$$

where $\alpha(\tau)$ is the behaviour resulting from the transition effects, exit, entry and do activities associated with $\tau$. $\iota(\ell_1)$ denotes the new basic control state which becomes part of the active configuration as a result of $\tau$’s execution.  

$\alpha(\tau)$ is specified as follows.

$$\alpha(\tau) = \alpha_{up}(\ell_0, lca(\ell_0, \ell_1), \varepsilon) \land \alpha_{down}(\ell_1, \varepsilon) \land \alpha_{do}(lca(\ell_0, \ell_1), \varepsilon)$$

The auxiliary functions $\alpha_{up}, \alpha_{down}, \alpha_{do}$ are specified as

$$\alpha_{up}(m_0, m_1, a) = \text{if } m_0 = m_1 \text{ then } a \text{ else } \alpha_{up}(p_s(m_0), m_1, a.exs(m_0))$$

$$\alpha_{down}(m, a) = \text{if } m \in BCS(s) \text{ then } a.en_s(m) \text{ else } \alpha_{down}(m, a.en_s(m), a')$$

Observe that $\tau'$ is well-defined in this definition, because $\omega(start(m))$ contains exactly one state machine transition.

$$\alpha_{do}(m, a) = \text{if } m = s \text{ then } a.do_s(m) \text{ else } \alpha_{do}(p_s(m), a.do_s(m))$$

Intuitively speaking, the composite behaviour associated with $\tau$ consists of

- the composition of exit actions, starting with the basic control state equal to or subordinate to $\ell_0$, going upward and ending at the direct substate of the least common ancestor $lca(\ell_0, \ell_1)$, and followed by
- the effect $\alpha$ associated with $\tau$, and then
- the effect of entering target state $\ell_1$.

If $\ell_1$ is a basic control state we are done. Otherwise we concatenate
• the effect of the transition from the start state which is a direct substate of $\ell_1$, to its target state, followed by

• the entry action of this target state,

and continue recursively from there until the basic control state associated with $\ell_1$ is reached.

The write set $W(\tau)$ associated with transition $\tau$ is equal to the write set of the behaviour $\alpha(\tau)$.

The new basic control state resulting from transition $\tau$ is $\iota(\ell_1)$, where $\iota$ is recursively defined as

$$\iota(m) = \begin{cases} m & \text{if } m \in BCS(s) \\ \text{let } \tau' = (\text{start}(m), e', g', \alpha', m') \in \omega(\text{start}(m)) \text{ in} \\ \iota(m') \end{cases}$$

Decomposition and Trigger of Discrete Transition Relation. Discrete transitions will be processed in several phases, these are reflected by mutually exclusive predicates $H_0, D_1, H_2$

$$\Phi_D \equiv (H_0 \lor D_1 \lor H_2)$$

An auxiliary variable $p \in \{0, 1, 2, 3\}$ selects $H_0, D_1, H_2, \Phi_T$ as will be seen below, therefore the trigger condition for any discrete transition to happen is that $p$ should be 0, 1 or 2:

$$\text{trigger}_D \equiv (p < 3)$$

For every event pool $q$ one event is concurrently dispatched. Boolean variables $b_q$ indicate whether a discrete transition could be transformed in pool $q$. Due to the potential existence of deferred events in the pool, the next event to be processed is not necessarily the head of the queue; its index is specified for each pool by variable $i_q$. 
**Housekeeping Phase $H_0$.** This predicate specifies how events are concurrently dispatched from event pools.

\[
H_0 \equiv p = 0 \land p' = 1 \land 0 \leq j \leq 1 \land \\
(\bigwedge_{s \in SM} \#\zeta(s)' = 0) \land (\bigwedge_{s \in SM} W'(s) = \emptyset) \land \\
\text{let } q = q_j \text{ in} \\
(\#q > 0 \land i_q \geq 0 \land q' = q \land i'_q = i_q \land e' = at(q, i_q) \land b'_q) \lor \\
(\#q > 0 \land i_q < 0 \land q' = q, \varepsilon_1 \land i'_q = \#q + 1 \land e' = \varepsilon_1 \land -b'_q) \lor \\
(\#q = 0 \land q' = \varepsilon_1 \land i'_q = 0 \land e' = \varepsilon_1) \land -b'_q) \land \\
(\bigwedge_{w \in V - \{(\zeta(s), W(s) | s \in SM) \cup \{q_0, q_1, d_q, b_q, p\}} w' = w) \\
\text{endlet}
\]

The following auxiliary variables are used in this predicate.

- $p$ — discrete transition processing phase identifier
- $j$ — index for the event pool to be processed in this phase sequence
  - $j = 0$: global event pool shared between TE and SUT
  - $j = 1$: local event pool inside SUT
- $e$ — the event to be dispatched
- $W(s)$ — write sets, one per state machine
- $\zeta(s)$ — output queue for intermediate storage of events created by state machine $s$ during a run-to-completion
- $b_q$ — flag (one per event pool $q_0, q_1$) indicating whether this event pool should be processed again, because it may contain further events
- $i_q$ — index (one per event pool $q_0, q_1$) indicating the next element from the pool to be processed.

Predicate $H_0$ applies if the phase $p$ is zero, and it defines phase 1 as its post-state. The housekeeping step specified by $H_0$ operates on the event pool identified by $j$. It clears the auxiliary output queues $\zeta(s)$ and write sets $W(s)$ of each state machine $s$. For event pool $q = q_j$ three possibilities have to be handled.

- The queue is non-empty and contains at least one event that can be processed in the current step. This event is selected at index position
Because the index is accordingly maintained as will become apparent in the definition of $D_1$. In this case the flag $b_q$ is already set to true because it is certain that this pool has to be scanned at least one more time for further events.

- The queue is non-empty, but all events in the queue are deferred and cannot be processed in the current step because it is already known that they do not trigger any transition. In this situation an $\varepsilon_1$ event is appended to the queue $q$, and this is also the next event to be processed. As a consequence, only transitions without user-defined events as triggers can be processed in the current step. Flag $b_q$ is set to false, because no further processing of the current pool is necessary, if no discrete step is triggered by $\varepsilon_1$. Therefore $b_q$ can be set to true in the next transition phase as specified by $D_1$.

- The queue is empty. Again, an $\varepsilon_1$ event is appended to the queue $q$.

All variables not changed in this discrete step $H_0$ keep their old values; this is specified by the last conjunct defining $H_0$.

**Discrete Transition Phase $D_1$.** Predicate $D_1$ specifies how an event selected during phase 0 as specified by $H_0$ is processed. The same auxiliary variables are used as introduced above for $H_0$.

$$D_1 \equiv p = 1 \land (p' = 2 \cdot j) \land j' = j + 1 \land (D_{11} \lor D_{12}) \land \left( \bigwedge_{v \in V - W'} v' = v \right)$$

From phase 1 we proceed to phase 2 if the SUT-internal event pool has been processed ($j=1$), but repeat phase 0 if $q_1$ is still to be processed. Predicates $D_{11}$ and $D_{12}$ distinguish the cases whether the current event $e$ triggered a transition or not. Both of these predicates specify a write set $W'$. All variables not contained in $W'$ remain unchanged.
\[ D_{11} \equiv \text{let } q = q_j \text{ in } \]
\[
\begin{align*}
&\left( \bigvee_{s \in SM, \tau \in \Sigma(s)} \text{trigger}_{s}(e, \tau) \right) \land \\
&\left( \bigwedge_{s \in SM, \tau \in \Sigma(s)} (\text{trigger}_{s}(e, \tau) \Rightarrow \epsilon(\tau)) \right) \land \\
&\left( \bigwedge_{s \in SM, \ell_0 \in \text{BCS}(s)} (\ell_0 \land \bigwedge_{\ell \in [\ell_0..s]} \neg \text{trigger}_{s}(e, \tau) \Rightarrow (\bigwedge_{\ell \in [\ell_0..s]} \epsilon(\text{do}_{s}(\ell)))) \right) \\
&W' = \{ q, i_{q_0}, i_{q_1}, b_q \} \cup \left( \bigcup_{\tau \in \{ \tau' | \exists s \in SM.s: \tau' \in \Sigma(s) \land \text{trigger}_{s}(e, \tau') \}} W(\tau) \right) \cup \\
&\left( \bigcup_{\ell \in \{ \ell' | \exists s \in SM.\ell_0 \in \text{BCS}(s) \land \ell_0 \land \ell' \land \ell' \in [\ell_0..s] \land \forall \ell'' \in [\ell_0..s] \land \tau \in \omega_{s}(\ell''): \neg \text{trigger}_{s}(e, \tau) \} \}} W(\text{do}_{s}(\ell)) \right) \land \\
&q' = \text{delete}(q, i_q) \{ \gamma(s)' | s \in SM \} \land b_q' \\
\end{align*}
\]
\end{verbatim}

The first conjunct in \( D_{11} \) asserts that at least one transition fires in presence of the dispatched event \( e \). The second conjunct states that – since concurrent behaviours triggered by the same event are executed in a concurrent way – the effect of all these behaviours becomes visible. Observe that all state machines are sequential, because we only allow non-orthogonal composite states. Therefore at most one transition per state machine fires.

The third conjunct specifies the effect of all do activities performed for active control states from where no transition is taken from any location \( \ell \) in the complete hierarchy from associated basic control state to top-level state \( s \).

The set \( W' \) is the union over all transition write sets and all write sets from do activities performed for control states specified by the third conjunct. Therefore it contains all variables written to during this discrete step.

Event \( e \) processed here is deleted from its pool. The events generated by all state machines involved are appended to the pool \( q \). We use the notation

\[ u\{.v \mid v \in X\} \]

to denote queue \( u \) with queues \( v, v \in X \) appended in arbitrary order, but without changing the order inside each \( v \).

Since we have processed a transition, the new index from where to read a new event from \( q \) is \( i_q' = 0 \): If there were deferred events in \( q \), and therefore
$i_q > 0$, these events have to be tried out again, since a transition has been performed. The same holds for the other pool, because the changes caused by performing transition $D_{11}$ may result in a guard condition to become true, so that a previously deferred event from that pool can now be dispatched. Therefore $i'_{q_0} = i'_{q_1} = 0$.

Predicate $D_{12}$ is applicable if no transition fires due to $e$.

\[
D_{12} \equiv \text{let } q = q_j \text{ in }
\]
\[
\left( \bigwedge_{s \in SM, \tau \in \Sigma(s)} \neg \text{trigger}_s(e, \tau) \right) \land
\left( \bigwedge_{s \in SM, \ell_0 \in BCS(s)} (\ell_0 \Rightarrow (\bigwedge_{i \in [0..s]} e(\ell))) \right)
\]
\[
W' = \{ q, i_q, b_q \} \cup \left( \bigcup_{\ell \in \{ \ell' | \exists s \in SM, \ell_0 \in BCS(s); \ell_0 \land \ell' \in [\ell_0..s] \}} W(\ell) \right)
\]
\[
(q' = \text{if } \delta_e \text{ then } q \text{ else delete}(q, i_q) \text{ endif}) \land
\]
\[
(i_q' = \text{if } \delta_e \land i_q < \#q - 1 \text{ then } i_q + 1 \text{ elseif } \delta_e \text{ then } -1 \text{ else } 0 \text{ endif}) \land
\]
\[
b_q' = b_q
\]

endlet

In this case no transition fires and the event is removed from the pool unless it is currently deferred. In the latter case $e$ remains in the pool at its current position.

The next index value of $i_q$ for accessing the pool is determined as follows.

- If there is another event in the pool that is not deferred it must be at position $i_q + 1$.
- If the pool is non-empty but only contains deferred events which cannot fire in the current configuration, $i_q'$ is set to -1. This indicates in the next phase 0 for the same pool that no suitable events are available though the pool is non-empty.

Even when no transitions fire, do activities may be performed in all control states which are part of the active configuration. This is reflected by the second conjunct and by the definition of $W'$.

**Housekeeping Phase $H_2$.** During the second housekeeping phase it is decided whether to continue with discrete steps ($p' = 0$) or to perform a delay
transition \((p' = 2)\).

\[
H_2 \equiv p = 2 \land \neg b'_{q_0} \land \neg b'_{q_1} \land j' = 0 \land \\
(\neg b'_{q_0} \land \neg b'_{q_1} \land p' = 3) \lor ((b_{q_0} \lor b_{q_1}) \land p' = 0) \land \\
\left(\bigwedge_{v \in V - \{p, b_{q_0}, b_{q_1}, j, \hat{t}\}} v' = v\right)
\]

This decision is made as follows.

- If at least from one pool a discrete transition could be triggered, we re-start with phase 0 and try out whether another event can be processed.
- Otherwise a delay transition is initiated; this is done by setting \(p' = 3\).

**Delay Transition Relation.** Delay transitions are specified by

\[
\Phi_T \equiv p = 3 \land p' = 0 \land i'_{q_0} = 0 \land i'_{q_1} = 0 \land \neg b'_{q_0} \land \neg b'_{q_1} \land j' = 0 \land \hat{t} > \hat{t} \land \\
\left(\bigwedge_{v \in V - (T \cup \{p, b_{q_0}, b_{q_1}, j, \hat{t}\})} v' = v\right) \land \\
(\forall s \in SM, (\ell_0, c, g, \alpha, \ell_1) \in \Sigma(s) : \\
(\exists \bar{g} \in Bexpr, t \in T, c \in \mathbb{N} : g \equiv \bar{g} \land \hat{t} - t \geq c) \Rightarrow (\hat{t}' \leq c + t \lor \hat{t} \geq c + t))
\]

This formula contains the following rules.

- The next phase to be processed will be phase 0 again, trying a novel discrete step on event pool \(q_0, (j' = 0)\), and trying to dispatch the first event contained in each queue \((i'_{q_0} = i'_{q_1} = 0)\).
- The model execution time is advanced.
- Input variables and the event pool between SUT and TE \((q_0)\) may change in the post-state of the delay transition, but all other variables and basic control states remain unchanged.
- The admissible time shift is limited by the point in time when the next timer will elapse. More precisely,
  - whenever a timer \(t\) is still running \((\text{so } \hat{t} - t \geq c = \text{false})\) the time may advance at most as far as the point in time where \(t\) will elapse, that is, \(c + t\). Equivalently, the new model execution time value \(\hat{t}'\) shall satisfy \(\hat{t}' \leq c + t\).
– Alternatively, \( t \) may have already elapsed before the delay transition is executed, that is, before current time \( \hat{t} \). This is characterised by condition \( \hat{t} \geq c + t \). In that case, timer \( t \) does not restrict the amount of time \( \hat{t} \) may be advanced.

11.2 Transition Relation for CML Interleaving Semantics

For constructing the transition relation of concurrent CML processes with the proper interleaving semantics we will use the classical approach described by Apt and Olderog [AO10, pp. 380]. The authors explain how guarded communication of concurrent “CSP-like” programs can be interpreted in an operational semantics: a communication

\[
P = \ldots \text{g}&c!x \rightarrow \ldots
\]
\[
Q = \ldots \text{h}&c?y \rightarrow \ldots
\]
\[
\text{SYSTEM} = P \deny \{c, \ldots \} \allow Q
\]

over channel \( c \) may take place if

– the execution state of both processes is at statements \( g&!x \) and \( h&c?y \), respectively, and,

– the communication guards \( g, h \) of the processes involved evaluate to true.

The effect of the communication can be described as the effect of the assignment \( y = x \);, which is specified as

\[
y = x\]

in a transition relation (“the post state of \( y \) after the channel communication equals the pre-state of \( x \) just before the channel communication took place”).

We illustrate the creation of a transition relation \( \Phi \) using the introductory example from the CML definition [WCC+12] whose original specification is shown in Fig. [11.4].

In a test the test environment (TE) has control over the channel communications taking place with the system under test (SUT) – see Fig. [11.5]. For inputs from TE to SUT, the test environment has also control over the values
functions
  oflow: (i,j:Byte) b: bool
  post b = (i+j > 255)

  uflow: (i,j:Byte) b: bool
  post b = (i-j < 0)

channels
  ...
I = { init, overflow, underflow, read, load, add, sub }

process RegisterProc =
  begin
    state reg: Byte
    initial
      INIT ()
      frame wr reg: Byte
      post reg = 0
    operations
      LOAD (i: Byte)
      frame wr reg: Byte
      post reg = i;

      READ () j: Byte
      frame rd reg: Byte
      post j = reg;

      ADD (i: Byte)
      frame wr reg: Byte
      pre not oflow(reg,i)
      post reg = reg^- + i;

      SUB (i: Byte)
      frame wr reg: Byte
      pre not oflow(reg,i)
      post reg = reg^- - i
    actions
    REG =
      load?i -> LOAD(i); REG
      [] dcl j: Byte @ j := READ(); read!j -> REG
      [] add?i -> ( oflow(reg,i) & overflow -> INIT(); REG 
      [] not oflow(reg,i) & ADD(i); REG )
      [] sub?i -> ( uflow(reg,i) & underflow -> INIT(); REG 
      [] not uflow(reg,i) & SUB(i); REG )
    @ init -> INIT(); REG
  end

Figure 11.4: CML specification RegisterProc process from WCC+12
Figure 11.5: Test scenario with environment TE and system under test REG to be passed along the channels. For each channel c between TE and SUT we therefore introduce auxiliary variables

- `<PROCESSNAME>_c` indicating that the CML process representing the SUT is willing to communicate over c.
- `TE_c` indicating that the test environment is willing to communicate over c.

These flags are of type Boolean if the processes involved have exactly one syntactic location where the communication over a channel c takes place. This is the case in the example from Fig. 11.4. Otherwise the flags become counters, a value `<PROCESSNAME>_c > 0` indicating that the process is willing to communicate over c at syntactic location number `<PROCESSNAME>_c`.

For the `RegisterProcess` example this leads to Boolean variables

```
REG_init, REG_load, REG_add, REG_sub,
REG_read, REG_overflow, REG_underflow,
TE_init, TE_load, TE_add, TE_sub,
TE_read, TE_overflow, TE_underflow,
```

Typically, test oracles will always listen to SUT output channels in order to detect transient SUT output errors. Therefore the Boolean flags `TE_c` associated with SUT outputs will always be true, and the TE implements test oracles as parallel processes never refusing to communicate over these channels.

The CML specification of the SUT process `REG` is now instrumented with `REG_c` variable assignments: `REG_c` is set to `true` (=1) if the SUT is in the execution state where the communication may occur and a guard condition – if any – associated with this communication evaluates to true.

Furthermore, the TE is equipped with channel variables for passing values to the SUT; if the SUT receives data over a channel in some variable x, then the TE is equipped with a variable `TE_x`.

For test generation purposes, the Boolean flags `TE_c` and variables `TE_x`, together with the test execution time `timeTick` are associated with version identifiers: `TE_c@n`, `TE_x@n`, `timeTick@n` is the n-th version of the respective
```plaintext
process RegisterProc =
begin
  ...
  actions
  REG =
  REG_init = 0; REG_load = 1; REG_add = 1; REG_sub = 1;
  REG_read = 1; REG_overflow = 0; REG_underflow = 0;
  (load?i -> LOAD(i); REG
   []
   var j: Byte @ j := READ(); read!j -> REG
   []
   add?i -> ( oflow(reg,i) & REG_init = 0; REG_load = 0; REG_add = 0;
             REG_sub = 0; REG_read = 0; REG_overflow = 1;
             REG_underflow = 0;
             overflow -> REG_overflow = 0;
             INIT(); REG
   []
   not oflow(reg,i) & ADD(i); REG )
   []
   sub?i -> ( uflow(reg,i) & REG_init = 0; REG_load = 0; REG_add = 0;
             REG_sub = 0; REG_read = 0; REG_overflow = 0;
             REG_underflow = 1;
             underflow ->
             REG_underflow = 0; INIT(); REG
   []
   not uflow(reg,i) & SUB(i); REG )
@ REG_init = 1; REG_load = 0; REG_add = 0; REG_sub = 0;
  REG_read = 0; REG_overflow = 0; REG_underflow = 0;
  init -> INIT(); REG
end
```

Figure 11.6: Instrumented version of `RegisterProc` process

variable, as calculated by the SMT solver in order to reach a certain test objective \( G \) while respecting the unrolled transition relation \( \Phi \).

The instrumented SUT process `REG` is shown in Fig. 11.6. Observe that the `REG_<channel>` flags are set to 1 if and only if the process is ready to communicate over `<channel>`. Further recall that choice

```plaintext
var j: Byte @ j := READ(); read!j -> REG
```

is ready to communicate over channel `read` at the same time when the process is ready to communicate over `load`, `add`, `sub`. Therefore `REG_read` is set at the same time as `REG_load`,.....,`REG_sub`. 

121
\[ I \equiv \]
\[
\text{REG\_init} = 1 \text{ and REG\_load} = 0 \text{ and REG\_add} = 0 \text{ and}
\]
\[
\text{REG\_sub} = 0 \text{ and REG\_read} = 1 \text{ and}
\]
\[
\text{REG\_overflow} = 0 \text{ and REG\_underflow} = 0
\]

Figure 11.7: The initial condition \( I \) associated with RegisterProc process.

Fig. [11.7] specifies the initial condition of a test: the SUT process REG may only communicate over channel init, therefore only REG\_init is 1. The internal state variables (here: variable \( \text{reg} \)) are undefined, therefore these variables do not occur in the predicate \( I \).

The transition relation is shown in Fig. [11.8] It is a disjunction structured according to the channel communications that may occur. For variable symbol \( x \), its pre-state is denoted by \( x\sim \), its post-state at the end of a transition execution by \( x \).

For example,

1. if both TE and SUT are willing to communicate over channel init – that is, if \((\text{TE\_init} \sim \text{ and REG\_init})\sim\) holds – this results in \( \text{reg} \) being set to 0 by the \( \text{INIT()} \)-function of REG and the communication flags set such that the TE may select communication over any of the channels load, add, sub, read.

2. if TE and SUT are willing to communicate over the add channel, the data value \( \text{TE\_i} \) passed by the TE to the SUT is copied to SUT variable \( i \). If the current value of the register plus \( i \) leads to an overflow the register remains unchanged and the SUT can only communicate further by issuing the \( \text{overflow} \) signal. Otherwise the value of \( i \) is added to the previous value of the register.

Observe that the sequential function and operation executions between two channel communications are represented in the transition relation by a single transition, because their execution is atomic and the execution of these steps occurs in zero time. If assignments to global attributes are made in the original SysML model, this would be modelled in CML by read/write channels operating on a process representing the global variables, so only local variable manipulations are “contracted” into a single transition step.
Φ ≡

(TE_init~ and REG_init~ and reg = 0 and
  REG_init = 0 and REG_load = 1 and REG_add = 1 and REG_sub = 1 and
  REG_read = 1 and REG_overflow = 0 and REG_underflow = 0 and
  j = reg)
  or

(TE_load~ and REG_load~ and i = TE_i~ and reg = i and
  REG_init = 0 and REG_load = 1 and REG_add = 1 and REG_sub = 1 and
  REG_read = 1 and REG_overflow = 0 and REG_underflow = 0 and
  j = reg)
  or

(TE_read~ and REG_read~ and TE_j = j~ and reg = reg~ and
  REG_init = 0 and REG_load = 1 and REG_add = 1 and REG_sub = 1 and
  REG_read = 1 and REG_overflow = 0 and REG_underflow = 0 and
  j = reg)
  or

(TE_add~ and REG_add~ and i = TE_i~ and
 (((reg~ + i > 255) and reg = reg~ and
   REG_init = 0 and REG_load = 0 and REG_add = 0 and REG_sub = 0 and
   REG_read = 0 and REG_overflow = 1 and REG_underflow = 0)
   or

 (not(reg~ + i > 255) and reg = reg~ + 1 and
   REG_init = 0 and REG_load = 1 and REG_add = 1 and REG_sub = 1
   and
   REG_read = 1 and REG_overflow = 0 and REG_underflow = 0 and
   j = reg)))
  or

(TE_overflow~ and REG_overflow~ and
  REG_init = 0 and REG_load = 1 and REG_add = 1 and REG_sub = 1 and
  REG_read = 1 and REG_overflow = 0 and REG_underflow = 0 and
  reg = 0 and j = reg))
  or

(TE_sub~ and REG_sub~ and i = TE_i~ and
 (((reg~ - i < 0) and reg = reg~ and
   REG_init = 0 and REG_load = 0 and REG_add = 0 and REG_sub = 0 and
   REG_read = 0 and REG_overflow = 0 and REG_underflow = 1)
   or

 (not(reg~ - i < 0) and reg = reg~ - i and
   REG_init = 0 and REG_load = 1 and REG_add = 1 and REG_sub = 1
   and
   REG_read = 1 and REG_overflow = 0 and REG_underflow = 0 and
   j = reg)))
  or

(TE_underflow~ and REG_underflow~ and
  REG_init = 0 and REG_load = 1 and REG_add = 1 and REG_sub = 1 and
  REG_read = 1 and REG_overflow = 0 and REG_underflow = 0 and
  reg = 0 and j = reg))

Figure 11.8: The transition relation Φ associated with RegisterProc process
Chapter 12

Automated Test Data Generation

Test data generation is performed by solving bounded model checking instances of the form discussed in the previous chapters, using an SMT solver. For solving an instance

\[ J(s_0) \land \bigwedge_{i=0}^{n} \Phi(s_i, s_{i+1}) \land G(s_0, \ldots, s_{n+1}) \]

the solver finds appropriate values of (see Chapter 11)

- SUT input variables (property values to be passed from TE to SUT),
- input events encoded as Boolean values,
- time increments when delay transitions are enabled,

for each model execution state \( s_i \), starting from an initial model state or from the last model state reached when creating test data for the previous symbolic test case. The test generator uses this data to create a test stimulator, that is, a thread of the test procedure used to pass input vectors to the SUT at appropriate points in time. A simulator is used to execute the data on the model, in order to find the last stable state reached in the run to completion when executing the input trace segment calculated by the solver. Observe that the solver itself cannot be used for this task: using cone of influence reduction, the solver usually operates with the transition relation of a sub-model which suffices to calculate the input data and delays needed to realise the test objective \( G \). The interpreter takes this data and calculates the resulting
trace and its final run to completion after the last input for the complete model.

12.1 SMT Solver

**SONOLAR.** Our SMT solver SONOLAR\(^1\) follows the bit blasting approach, variables are encoded as fixed-width bit vectors, where the bit widths are given by the associated data types. Arithmetic and logical operations on these variables are transformed to Boolean constraints that encode the exact relationship of input and output bits. This allows us to have bit-precise results in the presence of modular arithmetic.

To this end the SMT formula is first transformed to a directed acyclic formula graph, where each single arithmetic and logical operation is represented as a single node. Structural hashing ensures that structurally identical terms are shared among expressions. On this formula graph a series of word-level simplifications like the evaluation of constant expressions, normalizations and term rewriting is performed. This word-level formula graph is then transformed to a bit-level, purely propositional And-Inverter Graph (AIG). AIGs are commonly used among recent bit vector SMT solvers for synthesising propositional formulas [JLS09, Bru09, JSWD09]. AIGs represent propositional formulas as directed acyclic graphs (DAGs), where nodes are propositional variables or two-input AND-gates and edges may be optionally inverted. These AIG nodes are structurally hashed, too, and allow us to perform simplifications on bit level.

Although a number of competitive SAT solvers accept AIGs as input [Sör10, JC09], most SAT solvers require the input to be in CNF. To generate the CNF, for each node of the AIG a boolean variable is introduced. Each node with possibly inverted inputs \( n \leftrightarrow in_1 \land in_2 \) is then translated to \( (\neg n \lor in_1) \land (\neg n \lor in_2) \land (n \lor \neg in_1 \lor \neg in_2) \). For each root of the AIG an additional unit clause containing the associated variable asserts the corresponding boolean formula to be either true or false, respectively. See [EMS07] for more information on logic synthesis using AIGs.

Many modern SAT solvers have the capability to be called incrementally. This technique allows us to add clauses between solver runs and to add unit clauses that are only valid for one run (so-called assumptions). The SAT solver can

\(^1\)Solver for non-linear Arithmetic, see http://www.informatik.uni-bremen.de/~florian/sonolar/
then re-use conflict clauses learned in previous runs to speed up the following ones.

The solver tries to find a solution of a BMC instance in several steps, increasing the number $n$ of unrolled transition steps, starting with $n = 1$ (the solver is never activated if the goal $G$ is already fulfilled in $s_0$) and incrementing $n$ if no solution could be found. If $G$ can be determined on a state sequence of length $k < n$, it is always tried to solve $G$ on the last $k$ states of the trace. This means that the BMC instance is internally adapted to

$$J(s_0) \land \bigwedge_{i=0}^{n} \Phi(s_i, s_{i+1}) \land G(s_{n+2-k}, s_{n+3-k}, \ldots, s_{n+1})$$

with successively larger $n$.

Datatypes Supported by SONOLAR. SONOLAR supports Boolean, integer, and floating point data types. It implements a theory of arrays. This theory will be used for handling lists and sets in the CML: since test generation is always performed on a bounded number of transition steps, it can be assumed that all list or set-valued variables have bounded capacity. Therefore they can be implemented as ring buffers over arrays, using two additional integer variables acting as read and write index of the buffer.

12.2 Abstract Interpreter

An abstract interpreter has been developed to speed up the SMT solver. It is used to give a conservative estimate for the lowest bound $n$, for which a solution of a BMC instance can be found. Further details about SMT solver utilisation and its speed up by means of abstract interpretation can be found in [PVL11a].
Chapter 13

Test Model V&V By Bounded Model Checking

The crucial role of the test model in model-based testing suggests that support for model verification and validation should be made directly available in the test automation tool. Since RTT-MBT solves BMC instances for the purpose of test data generation anyway, it is an obvious choice to use bounded model checking techniques for this purpose.

13.1 Bug Finding By Bounded Model Checking

In [BHJ+06] and [CBRZ01] the authors explain how model checking of LTL formulas can be performed by transforming them into BMC instances. It is described how BMC can even prove global properties of the model, if the transition relation is unrolled until the diameter of the model’s underlying transition graph is reached. This diameter, however, will be far too large in the context of SoS testing, even when using adequately abstracted models. As a consequence we offer BMC only for the purpose of bug finding, that is, for finding unwanted system states in the vicinity of a model state, each potential error state reachable by a bounded number of transitions.
13.2 BMC Against SafetyLTL Formulas

As described in Chapter 9, specifying properties of finite traces can be suitably performed using the LTL subset called SafetyLTL. The technique explained there is suitable for specifying symbolic test cases, but for the purpose of bug finding it has to be extended with respect to the $G$ operator: while $G$ can be approximated for test case specification by finite applications of the $X$ operator, it is desirable to find cyclic paths in the vicinity of a start state $s_0$, so that an unwanted property of the form $G\phi$ (for example, a livelock) can be detected on this cycle.

The authors of [BHJ+06] and [CBRZ01] describe how to identify so-called lasso shaped paths starting from $s_0$: these paths consist of a start trace segment $\beta = s_0 \ldots s_{\ell-1}$ followed by a finite cycle segment $\gamma = s_{\ell} \ldots s_n$ satisfying $\Phi(s_n, s_{\ell})$, so that $\beta.\gamma^\omega$ (that is, $\beta$ followed by an infinite repetition of $\gamma$) is a computation of the model (Fig. 13.1).

If the initial trace segment $\beta$ to this cycle and all nodes on the cycle $\gamma$ itself fulfil $\phi$, a solution of $G\phi$ has been found. The identification of lasso shaped paths and, to this end, of recurring states already visited while unrolling the transition relation, is not needed for test generation, but is required for the purpose of bug finding.
Part III

COMPASS Platform Integration
Chapter 14

Eclipse User Interface Plugin for RT-Tester

The Rtt-Mbt plugin is part of the COMPASS tool integration in the Eclipse framework. It provides access to Rtt-Mbt testing functionality for the COMPASS SoS verification and validation work flow. This chapter describes the Architecture, structure and information flow of this part of the COMPASS tool development.

14.1 Architecture

While the Rtt-Mbt implementation currently has specific platform requirements and can benefit from high CPU speed and the amount of available memory, the COMPASS integration in the Eclipse framework is designed to run on as many platforms and machines as possible. To resolve this situation, a server component has been implemented as part of the Rtt-Mbt tool suite that supports remote execution of Rtt-Mbt functionality. A Rtt-Mbt Java API implements a client for the server component that provides the Rtt-Mbt functionality as Java API calls to the actual COMPASS Rtt-Mbt plugin. The structure of this architecture is depicted in Fig. 14.1.

The Rtt-Mbt Java API client connects to the Rtt-Mbt server through TCP/IP connections. The protocol of JSON objects between client and server is described in chapter 15. The architecture of the Java client API is described in the following section 14.2. It provides a central Java class for all Rtt-Mbt tasks. The Rtt-Mbt COMPASS plugin implements the user interface to
these RTT-MBT tasks and the RTT-MBT work flow. The architecture of the plugin is described in section 14.3.

The COMPASS CML tool Eclipse Plugin works with CML projects that contain the different components of a COMPASS project. Generating test procedures with RTT-MBT always requires a test model that describes the behaviour of the system under test (SUT) and the test environment (TE) in which the tests of a SUT are executed. From these two definitions, a set of tests can be generated that can test different parts of the SUT. The test model and the tests for this model are handled together in a RTT-MBT component within a CML project.

14.2 RTT-MBT Java API

The RTT-MBT Java API is implemented in a package rttMbtTmsClientApi and integrated in the COMPASS CML tool build environment\(^1\). It is imported by the RTT-MBT Eclipse plugin and provides client functionality for RTT-MBT commands. The class hierarchy of this package is presented in Fig. 14.2. To improve readability of the figure, only the class attributes and functions are listed, that are explained in this section.

The class RttMbtClient provides functions for all RTT-MBT tasks that the plugin classes can perform on the RTT-MBT server. The complete work flow from generating a RTT-MBT component (createProject), initialising the component (initProject), generating test procedures and sim-

\(^1\) For details about the build environment please refer to [CML+13].
ulations (generateTestProcedure, generateSimulation) as well as compiling, executing, documenting and cleaning up a generated test procedure (compileTestProcedure, runTestProcedure, docTestProcedure, cleanTestProcedure) is supported. In addition to these high level tasks, some lower level actions are supported to handle the remote working area of a client. The working area can be created (beginRttMbtSession) and removed (removeRttMbtSession) and files can be uploaded to the server (uploadFile, uploadDirectory) and downloaded from the server (downloadFile, downloadDirectory).

The class RttMbtClient uses the attributes log and progress to output text messages and progress indications that are sent by the server during the execution of a RTT-MBT task. The objects that are assigned to these attributes must implement the interfaces IRttMbtLoggingFacility and IRttMbtProgressBar. Classes that implement these interfaces are defined in the actual plugin implementation described in section 14.3.

The protocol between the RTT-MBT Eclipse plugin and the RTT-MBT server component is a sequence of JSON command and reply objects and is described in detail in chapter 15. When generating an instance of class RttMbtClient, the address and TCP port of the server has to be provided along with the user identification. The class RttMbtClient instantiates objects of class jsonCommand to perform the TCP/IP communication to the server. This class is the base class for all supported RTT-MBT client/server commands. For each command, a separate class is implemented that creates the respective JSON objects and handles the reply from the server. The server expects a separate TCP/IP connection for each action and drops this connection after the reply is sent successfully. The base class jsonCommand implements the functionality for the handling of the TCP/IP connection from connecting to the server (connectToServer), sending client information (sendClientInformation) and optionally the request for debug information (sendDebugInformation) to receiving the reply (receiveReply) and closing the connection (closeConnection). While receiving the reply, jsonCommand objects scan the received JSON objects for text messages from the server (scanForConsoleItems) and progress information (scanForProgressItems). Through the client attribute of the jsonCommand objects, the text messages and progress information can be sent to the IRttMbtLoggingFacility and IRttMbtProgressBar attributes of the RttMbtClient.

Refer to 15.1.2 for details about the TCP/IP communication behaviour.
Figure 14.2: Overview of the RTT-MBT Java client API classes.
14.3 RTT-Mbt Eclipse Plugin

The Eclipse plugin for RTT-Mbt components provides a user interface for the RTT-Mbt test generation and test execution work flow within the COMPASS CML tool Eclipse integration. It is implemented in the package `eu.compassresearch.ide.cml.rtt-plugin` and uses the functionality provided by the RTT-Mbt Java API to perform the respective RTT-Mbt tasks. The class hierarchy of this package is presented in Fig. 14.3.

The class `Activator` is created when the plugin is activated in the Eclipse framework. The `RttMbtClient` attribute of the class can be used by other classes of the plugin to perform RTT-Mbt commands. Separate view classes `RttMbtConsoleView` and `RttMbtProgressView` use implementations of `IRttMbtLoggingFacility` and `IRttMbtProgressBar` (`RttMbtConsoleLogger` and `RttMbtProgressBar`) to present text messages and progress information to the user.

Currently the plugin implementation provides a wizard to create a new RTT-Mbt component (`RttMbtComponentWizard`) and popup menu actions to perform the RTT-Mbt work flow actions within a RTT-Mbt component. The base class for all popup menu actions is `RttMbtPopupMenuAction`. This class implements functions to initialise the `RttMbtClient` object that is used to perform the action (`initClient`) and to get the currently selected object from the project view of the COMPASS perspective. Base classes for test generation context commands (`RttMbtAbstractTestProcedureAction`) and concrete test procedure commands (`RttMbtConcreteTestProcedureAction`) are inherited from `RttMbtPopupMenuAction`. They implement functionality to check if a selected project view object is a valid test generation context directory (`isTestProcedureSelected`) or concrete test procedure directory (`isRttTestProcedureSelected`). All popup menu actions have to implement the method `execute`.

Further plugin classes that provide UI functionality to configure the test procedure generation context and to analyse the test execution results are in development, but are not included in the RTT-Mbt Eclipse plugin, yet.
Figure 14.3: The RTT-MBT plugin classes.
Chapter 15

The JSON Protocol for Interaction Between Tool Components

15.1 Structure

The RTT-MBT Tool component can be used from the COMPASS tool by means of a TCP/IP client/server connection. The RTT-MBT server accepts TCP/IP connections from the COMPASS tool on a predefined port. This chapter describes the protocol that is supported by the RTT-MBT server component.

15.1.1 JSON Encapsulation

Each action that is to be performed by the RTT-MBT server is triggered by a sequence of command words that are sent to the server. The server sends a sequence of reply words. The commands and replies are JSON objects. Binary and string content is encoded in base64 format to avoid characters that are not allowed in JSON.

15.1.2 Command Sequence

A new TCP/IP connection is established for each action and is disconnected after the last reply object has successfully been received by the client. An
action should start with a client information object (see 15.2.1). The client information contains a unique id that is used to distinguish between the working directories of different clients.

The client information can be followed by debugging or compression information objects. These can be used to control the generation of debug information or the compression of the base64 encoded content.

The action that is to be performed is sent as a separate command object. There are JSON objects defined for each action that is supported by the Rtt-Mbt server. Higher level tasks can consist of several lower level Rtt-Mbt actions. E.g. the task to check that a model is live-lock free requires to store the model in a directory on the server side and to run the live-lock check. Storing the model and checking for potential live-locks are two separate Rtt-Mbt actions.

For each action that is successfully received by the Rtt-Mbt server, a job acknowledge object is generated and sent back to the client. If a result object is defined for the action, the result is sent to the client, followed by a debugging information object if debug information was requested. If errors occur, the list of all error messages are sent in an exception object after the result.

15.2 Session management

15.2.1 Client Identification

The client-information object is sent at the beginning of each action to identify the client that issues the action. The object contains values for user and user-id attributes. user is the human readable name of the user that is issuing the command and user-id is a unique id that identifies the user. The Rtt-Mbt server uses the user id to identify the working directories of a client. A special remove-workingdirs action can be used to cleanup all data for a specified client.

{"client-information":
{"user":"<user name on client>",
"user-id":"<unique user identification>"}}
15.2.2 Job Acknowledge

Every command object that is received and successfully parsed by the RTT-MBT server is acknowledged with a job-acknowledge object that contains the job identification number of the command. This job identification number is needed to abort a running command using the abort-command described in 15.2.5.

{"job-acknowledge":{"job-id":<job-identification-number>}}

15.2.3 Testing Connections

A client can test if an RTT-MBT server is active and reachable with the used IP address and port settings by trying to send a test-connection-command object.

{"test-connection-command":""}

If the connection to the port fails, no RTT-MBT server is active or it is not reachable. If sending the test-connection-command succeeds, a job-acknowledge object should be received followed by a test-connection-result object containing information about the server version and its up-time.

{"test-connection-result":
{"server-version":"<server version>",
"uptime":"<uptime>"}}

15.2.4 Retrieve Status Information

If a command results in an RTT-MBT task that takes some time, clients can check for the status of the task using get-job-status commands.

{"get-job-status":""}

Sending a get-job-status command should be answered with a job-acknowledge followed by a job-status-overview that contains information about all jobs of a user that are currently active. For each job, the name of the command, the job identification number, the user information and additional information about the status of the job is given.
Note that this requires that a client-information object has been sent prior to the get-job-status command on the same connection so that the RTT-MBT server can retrieve the user identification from it.

```
{ "job-status-overview":
  { "job-status" : <job-status-list> }
}
```

```
<job-status-list> ::= [<entry>,<additional entries>]
```

```
<entry> ::= {
  "job" : "<name-of-job>",
  "job-id" : <job-identification-number>,
  "user" : "<name of user that started the job>",
  "user-id" : "<user identification>",
  "info" : "<additional information>" }
```

### 15.2.5 Aborting Commands

Clients can abort a job using the `abort-command` and the job identification number of the respective job. The abort command has to be sent on a separate connection and not on the connection that should still be open for the job that is to be aborted. There is no result object for a successful termination of a task sent to the client on the connection on which the abort command is sent. On the connection on which the original command was sent, an exception object (see 15.10.1) is sent to the client containing the error message "command execution aborted".

If a job identification number is provided for which no job exists on the server, an exception object is returned on the connection of the `abort-command` containing the error message "unable to find job!".

```
{ "abort-command" : 
  { "abort" : <job-identification-number> } }
```

### 15.2.6 Cleanup

At the end of a working session, all working directories of a user should be removed to save disc space on the RTT-MBT server. This can be performed using a `remove-workingdirs-command` object with the respective user name.
and user identification information that is sent to the server. No result object is defined for this command.

"remove-workingdirs-command":
  {"user":"<user name on client>",
   "user-id":"<unique user identification>"}

15.3 Test Model

The test model is the central component for all RTT-MBT test generation tasks and is normally reused for different test generation context configurations of a test project. The RTT-MBT server provides a model store that supports this reuse of test models.

15.3.1 Storing Models

A model can be added to the model store on the server using a store-model-command object. The model-name and model-version values of the object identify the model in the store. The complete XMI export of the model must be added as a base64 encoded string as the model value of the store-model-command object. No result object is defined for this command. If a problem occurs, an exception object with the respective error message is sent to the client. Otherwise the connection is closed by the server after the job-acknowledge object has been sent successfully.

"store-model-command":
  {"model-name":"<name of the model>",
   "model-version":"<version string>",
   "model":"<base64 encoded content of XMI export>"}

15.3.2 Checking Model Files

Before sending the complete model to the server, a client can check if a specific model already exists in the store. The check-model-file-command contains values for the model name and version to identify the model and a SHA256 checksum that can be used to check if a model that is found in the store is identical to the one that is being checked.
D34.1 - Test Automation Support (Public)

{"check-model-file-command": {
  "model-name": "<name of the model>",
  "model-version": "<version string>",
  "model-file-checksum": "<SHA-256 checksum of the file>"}}

The RTT-MBT server returns a check-model-file-result object with the check result.

{"check-model-file-result": {
  "model-name": "<name of the model>",
  "model-version": "<version string>",
  "result": "<result string: PASS|FAIL>"}}

15.3.3 Live-lock Checking

Live-locks are not allowed in test models and can lead to errors in the test generation process. check-model-command objects can be used to command the RTT-MBT server to perform a live-lock check on the specified model. The model is identified by the model-name and model-version values of the object. Additional parameters can be added to control the live-lock check. If progress-port or console-port values are given, the server will send progress item or console item objects to the client before the final model-checking-results object is sent.

{"check-model-command": {
  "model-name": "<name of the model>",
  "model-version": "<version string>",
  <check-optional-parameters>}}

<check-optional-parameters> ::= "max-length":<number of maximal length a cycle/trail may have> , "log-extent": "<one of the keywords TINY, STANDARD, VERBOSE>" , "progress-port": "true" , "console-port": "true"

The result of a live-lock check is a live-lock report that is sent as a model-checking-results object to the client.

{"model-checking-results": "<base64 encoded check results>"}

\[\text{1\textsuperscript{st} see section 15.9 for details on progress and console items}\]

141
15.4 Test Generation Context

The test generation context defines how a concrete test procedure is generated, which part of the model should be covered by the concrete test procedure and which signal value ranges should be used during the generation. Empty templates for the configuration files can be generated from the test model using RTT-MBT commands.

15.4.1 Configuration

An empty configuration file `configuration.csv` that controls the model coverage of a generated test procedure can be created using `conftool-command` objects. The object must specify the model for which the `configuration.csv` file is to be generated.

```json
"conftool-command":
  {"model-name": "<name of the model>",
   "model-version": "<version string>"}
```

The server returns a `conftool-result` object that contains the content of the created `configuration.csv` file as a base64 encoded string.

```json
"conftool-result":
  {"configuration.csv": "<base64 encoded file content>"}
```

15.4.2 Signalmap

The value ranges for signals that are used during a test procedure generation are defined in the `signalmap.csv` file in the test procedure generation context. `sigmaptool-command` objects can be used to create a `signalmap.csv` file with undefined ranges for all signals. The object must specify the model in its `model-name` and `model-version`.

```json
"sigmaptool-command":
  {"model-name": "<name of the model>",
   "model-version": "<version string>"}
```

The `sigmaptool-result` object that is sent from the server to the client after a successful execution contains the base64 encoded content of the `signalmap.csv` file.
15.5 Test Procedure Generation

Generating a test procedure on the RTT-MBT server requires to have a RTT-MBT component directory structure transferred to a server working area of the client that initiates the test procedure generation. Working areas are created, maintained and destroyed using the commands described in section 15.8. For this section we assume that the working area has been created and is synchronised.

15.5.1 Generating Test Procedures

The concrete test procedure for a specific test procedure generation context is generated by the RTT-MBT server after a `generate-test-command` object is received. A test procedure generation context is always part of a RTT-MBT component. The generation context is identified by the name of the RTT-MBT component and its path within the RTT-MBT component.

```json
{"generate-test-command":
  "project-name": "<name of the RTT-MBT component>",
  "test-procedure-path": "<path-of-test-procedure-generation-context>"
  <tgen-optional-parameters>}}
```

Optional parameters can be provided to control the test procedure generation. The maximum number of SMT solver steps and simulation steps can be defined that are used by the generator. If a goal cannot be reached within these steps, the solver does not try to solve this goal any further and proceeds with the next goal.

If `progress-port` or `console-port` values are given, the server will send progress item or console item objects to the client before the `test-generation-results` object is sent.

```json
<tgen-optional-parameters> :=
  ,"max-solver-steps": "<max. SMT solver steps>"
  ,"max-simulation-steps": "<max. Simulation Steps>"
  ,"progress-port": "true"
  ,"console-port": "true"
```
A **test-generation-results** object is sent to the client after the generation process is complete. It only contains the information if the generation was successful or not. The generated files have been created in the server working area of the client and have to be transferred to the client using the commands described in section [15.8](#).

```json
{"test-generation-results":
  {"result":"<result string: PASS|FAIL>"}}
```

### 15.5.2 Generating Simulations

If a SUT or parts of a SUT should be simulated, **generate-simulation-command** objects can be used to start a simulation in the server working area of a client on the RTT-MBT server. A separate test procedure configuration context is used to define which parts of the test model are to be simulated.

```json
{"generate-simulation-command":
  {"project-name":"<name of the RTT-MBT>",
   "test-procedure-path":"<path-of-procedure-generation-context>"
   <generate-simulation-optional-parameters>}}
```

The same optional parameters as for **generate-test-command** objects apply to **generate-simulation-command** objects.

```json
<generate-simulation-optional-parameters> :=
  ,"max-solver-steps":"<max. SMT solver steps>"
  ,"max-simulation-steps":"<max. Simulation Steps>"
  ,"progress-port":"true"
  ,"console-port":"true"
```

A **simulation-generation-results** object is sent to the client after the generation process is complete. It only contains the information if the generation was successful or not. The generated files have been created in the server working area of the client and have to be transferred to the client using the commands described in section [15.8](#).

```json
{"simulation-generation-results":
  {"result":"<result string: PASS|FAIL>"}}
```
15.6 Test Procedure Commands

A concrete test procedure in an RTT-MBT component is named after the test generation context that has been used to generate it. A test procedure and its generation context can be distinguished by their path within the RTT-MBT component.

Compilation, execution and documentation of a test procedure on the RTT-MBT server requires to have the RTT-MBT component directory structure transferred to the server working area of the client that initiates the command. For this section we assume that the working area has been created and is synchronised.

15.6.1 Compilation

In the COMPASS project RTT-MBT is used to generate RT-Tester test procedures. Before an RT-Tester test procedure can be executed, it has to be compiled to generate the test procedure executable. The `compile-test-command` object can be used to trigger the compilation of test procedures on the RTT-MBT server. The test procedure must be identified by the name of its RTT-MBT component and its path within the component.

```json
{"compile-test-command":
  {"project-name"":"<name of the RTT-MBT component>",
   "test-procedure-path"":"<path-of-test-procedure>"}}
```

A `compile-test-result` object is sent to the client after the compilation is finished. It only contains the result of the compilation (PASS or FAIL). The test procedure executable does not have to be transferred to the client because it will be executed on the RTT-MBT server and not on the client side.

```json
{"compile-test-result":
  {"result"":"<result string: PASS|FAIL>"}}
```

15.6.2 Test Procedure Execution

After a test procedure has been successfully compiled, the generated test procedure executable can be executed in the RTT-MBT server working area of a client using a `run-test-command` object sent to the server. The test
procedure must be identified by the name of its RTT-MBT component and its path within the component.

```json
"run-test-command":
{"project-name":"<name of the RTT-MBT component>",
"test-procedure-path":"<path-of-test-procedure>"}
```

After the execution is finished, a `run-test-result` object is sent to the client with the result of the execution. Because the test data that is generated during the test execution can be very different from one test procedure to another, it is in the responsibility of the client to retrieve the needed files from the working area on the server. The commands described in section 15.8 are suitable for this task.

```json
"run-test-result":
{"result":"<result string: PASS|FAIL>"}
```

### 15.6.3 Documentation

RT-Tester supports automatic test procedure documentation generation. The documentation of a test procedure can be generated by sending an appropriately defined `doc-test-command` object to the server. The test procedure must be identified by the name of its RTT-MBT component and its path within the component.

```json
"doc-test-command":
{"project-name":"<name of the RTT-MBT component>",
"test-procedure-path":"<path-of-test-procedure>"}
```

When the documentation has been generated, a `doc-test-result` object is sent to the client with the generation result (PASS or FAIL). The generated documentation still resides in the working area of the client on the RTT-MBT server and has to be transferred to the client in separate commands.

```json
"doc-test-result":
{"result":"<result string: PASS|FAIL>"}
```

### 15.7 Replay Test Results

Replaying a test execution result is an important part of verifying the test result. The tool qualification if the RTT-MBT tool suite relies on the correct-
ness of the replay component and the fact that errors of the test generator will be detected by a replay of the test execution results.

Replaying the execution result of a test procedure on the RTT-MBT server requires to have the RTT-MBT component directory structure transferred to the server working area of the client that initiates the replay. For this section we assume that the working area has been created and is synchronised.

### 15.7.1 Replay

A replay is initiated by sending a `replay-command` to the server that identifies the generation context of the test procedure by the name of the RTT-MBT component and the path to the generation context within the component. The HTML test execution log is provided as a base64 encoded string value within the `replay-command` object. Additional parameters can be used to enable user feedback (progress indications and console messages).

```json
{ "replay-command": {
  "project-name": "<name of the RTT-MBT component>",
  "test-procedure-path": "<path-of-procedure-generation-context>",
  "replay": "<base64 encoded html-test execution-log"

  <replay-optional-parameters>}}

<replay-optional-parameters> :=
  , "progress-port": "true"
  , "console-port": "true"
```

The `replay-results` object that is sent to the client only contains the information if the replay was successfully performed or not. Any output files generated during the replay have to be transferred to the client with separate commands.

```json
{ "replay-results" :
  { "result": "<result string: PASS|FAIL>"}}
```

### 15.8 RTT-MBT Server Working Areas

Some of the RTT-MBT server commands do not generate output files or the output files are included in the result JSON objects. For tasks that generate a lot of output files or possibly very large output files, it is not recommended to include all result files in on result JSON object. In some cases not all files need
to be transferred from server to client, but may be interesting for debugging later. For these tasks, the concept of working areas on the RTT-MBT server has been developed. Working areas can be generated for each client and are identified by the client id as used in the client-information command described in section 15.2.1.

There is a difference between temporary working directories of single commands and working areas for a client. A single command normally is performed in a temporary working directory that is only used for this command and that normally is removed at the end of a working session using the remove-workingdirs-command described in [15.2.6]. Commands like the ones described in [15.5] and [15.6] work in the client working area on the server and do not create temporary working directories. The working area of a client is not removed by the remove-workingdirs-command so that a working area can be persistent for more than one session.

Commands that use working areas normally are part of a more complex action or task that consists of several commands like sending the input files to the work area on the RTT-MBT server, performing the actual command and retrieving selected output files generated or changed by the command.

A work area is always assigned to a user identification. The same user will only get one working area as long as the same user identification is used even if he connects from different clients. The user can work with different RTT-MBT components for different test models in the same working area, because the RTT-MBT component name is always used as a prefix of the path within the work area. If a user works in parallel with two different clients and the same user identification on the same RTT-MBT component, conflicts can occur.

15.8.1 Creating Work Areas

The work for a client on the RTT-MBT server has to be explicitly created by the client using a start-file-cache-command JSON command object. The user name and user identification has to be defined in the object.

{"start-file-cache-command":
"user":"<user name on client>",
"user-id":"<unique user identification>"}

The result of the work area generation command is sent to the client in form of a start-file-cache-reply object. If a working area for this client
identification already exists, the work area will be initialised and the result will be PASS.

{"start-file-cache-reply":
  {"user":"<user name on client>",
   "user-id":"<unique user identification>",
   "result":"<result string: PASS|FAIL>"}}

### 15.8.2 Destroying Work Areas

If a work area is no longer needed, it has to be explicitly destroyed by the client using a remove-file-cache-command object specifying the user identification of the work area that is to be removed.

{"remove-file-cache-command":
  {"user":"<user name on client>",
   "user-id":"<unique user identification>"}}

A remove-file-cache-reply object is sent to the client with the result of the command.

{"remove-file-cache-reply":
  {"user":"<user name on client>",
   "user-id":"<unique user identification>",
   "result":"<result string: PASS|FAIL>"}}

### 15.8.3 Checking For A Work Area

Before creating a work area, it can be useful to check if it already exists. If a check-file-cache-exists-command is sent to the Rtt-Mbt server, the server checks if a work area already exists for the specified user identification.

{"check-file-cache-exists-command":
  {"user":"<user name on client>",
   "user-id":"<unique user identification>"}}

The result of the check is sent to the client as a check-file-cache-exists-reply object.

{"check-file-cache-exists-reply":
  {"user":"<user name on client>",
   "user-id":"<unique user identification>",
   "result":<result string: PASS|FAIL>"}}
15.8.4 Send Files To Work Areas

Files from the local file system of the client can be sent to the work area on the server using a `send-file-to-cache-command` object. In addition to the user identification, the object must contain the file name, the size and a SHA256 checksum of the file to be sent as well as the base64 encoded content of the file.

Note that the file name has to be base64 encoded to avoid illegal characters and that it can contain path information. The path information is always interpreted relative to the root directory of the work area.

```json
{"send-file-to-cache-command":
   {"user":"<user name on client>",
    "user-id":"<unique user identification>",
    "filename":"<filename with path relative to cache root (base64)>",
    "content":"<base64 encoded content of the file>",
    "size":"<the size of the uncompressed file in bytes>",
    "checksum":"<SHA-256 checksum of the file>"}}
```

The server sends a `send-file-to-cache-reply` command to indicate whether the file has been successfully received and stored or not.

```json
{"send-file-to-cache-reply":
   {"user":"<user name on client>",
    "user-id":"<unique user identification>",
    "filename":"<filename with path relative to cache root (base64)>",
    "result":"<result string: PASS|FAIL>"}}
```

15.8.5 Check Files In Work Areas

Before sending files to the work area, it is recommended to check if the same file is already stored and does not need to be transferred again. A `check-file-in-cache-command` object that is sent to the RTT-MBT server triggers such a check. The file name and the SHA256 checksum of the file content has to be provided.

```json
{"check-file-in-cache-command":
```
If the file exists and is identical, a check-file-in-cache-reply object is sent with the result PASS. Otherwise the result coded in the object will be FAIL.

```
{"check-file-in-cache-reply":
  {"user": "<user name on client>",
   "user-id": "<unique user identification>",
   "filename": "<filename with path relative to cache root (base64)>",
   "result": "<result string: PASS|FAIL>"}}
```

### 15.8.6 Retrieve Files From Work Areas

To receive a file from a work area on the RTT-MBT server, the file name (and path) must be provided in the respective command object. Because the client cannot know in all cases which files exist in the work area, two commands support to retrieve lists of directories and files in a directory of the work area. The distinction between files and directories is necessary, because only a list of names is returned.

get-cache-directory-list-command objects can be sent to the RTT-MBT server to retrieve a list of directories in a directory of the work area.

```
{"get-cache-directory-list-command":
  {"user": "<user name on client>",
   "user-id": "<unique user identification>",
   "directory": "<directory path relative to cache root (base64)>"}}
```

The server will return a get-cache-directory-list-reply object containing the list of all directory names (without path information) within the specified directory in the work area. If the specified directory does not exist, the list is empty and an exception object with the respective error message is sent to the client.

```
{"get-cache-directory-list-reply":
  {"user": "<user name on client>",
   "user-id": "<unique user identification>",
   "directory-list": ["<dirname 1>",...,"<dirname n>"]}}
```
get-cache-file-list-command objects can be sent to the RTT-MBT server to retrieve a list of files in a directory of the work area.

```
{"get-cache-file-list-command":
  {"user":"<user name on client>",
   "user-id":"<unique user identification>",
   "directory":"<directory path relative to cache root (base64)>"}}
```

The server will return a get-cache-file-list-reply object containing the list of all file names (without path information) within the specified directory in the work area. If the specified directory does not exist, the list is empty and an exception object with the respective error message is sent to the client.

```
{"get-cache-file-list-reply":
  {"user":"<user name on client>",
   "user-id":"<unique user identification>",
   "file-list":["<filename 1>",...,"<filename n>"],
   "directory":"<directory path relative to cache root (base64)>"}}
```

Files that have been generated or altered by the RTT-MBT tool on the server can be retrieved by sending receive-file-from-cache-command objects to the server. The object must specify the work area and the file name including the path within the work area.

```
{"receive-file-from-cache-command":
  {"user":"<user name on client>",
   "user-id":"<unique user identification>",
   "filename":"<filename with path relative to cache root (base64)>"}}
```

The receive-file-from-cache-reply object that is sent by the server contains information about the size and the checksum of the file as well as the base64 encoded content of the file. The size and checksum enables the client to verify that the file has been transferred correctly.

```
{"receive-file-from-cache-reply":
  {"user":"<user name on client>",
   "user-id":"<unique user identification>",
   "filename":"<filename with path relative to cache root (base64)>",
   "content":"<base64 encoded content of the file>",
   "size":"<the size of the uncompressed file in bytes>",
   "checksum":"<SHA-256 checksum of the file>"}}
```
15.8.7 Remove Files In Work Areas

To remove a file in the work area on the server, a client sends a \texttt{remove-file-from-cache-command} object that specifies the work area and the file name including the path within the work area.

\begin{verbatim}
{"remove-file-from-cache-command":
{"user": "<user name on client>",
 "user-id": "<unique user identification>",
 "filename": "<filename with path relative to cache root (base64)>"}
\end{verbatim}

A \texttt{remove-file-from-cache-reply} object is sent by the server to indicate whether the removal was successful or not.

\begin{verbatim}
{"remove-file-from-cache-reply":
{"user": "<user name on client>",
 "user-id": "<unique user identification>",
 "filename": "<filename with path relative to cache root (base64)>", 
 "result": "<result string: PASS|FAIL>"}
\end{verbatim}

15.9 User Feedback

Some of the Rtt-Mbt tools perform time consuming tasks and can send progress data or can provide text based feedback to the user. This feedback as well as information about the progress of a task can be sent to clients, if requested.

15.9.1 Progress Indication

The progress of Rtt-Mbt tasks like live-lock checking or test generation can be requested in optional parameters of the respective JSON command objects. If requested, the server will send \texttt{progress-item} objects before the actual result object to the client.

\begin{verbatim}
{"progress-item": "<content of progress port datagram>"}
\end{verbatim}

An Rtt-Mbt command can sometimes consist of several sub tasks. Progress items can be generated for each sub task. They carry a string that contains the name of the sub task and a progress (in percent) of the sub task. Checking a model for potential live-locks is a Rtt-Mbt task that is not divided into sub tasks. A progress indication that 37 percent of the...
check have been performed would be indicated by a progress item value "Check Model: 37".

15.9.2 Console Output

Text based indication of RTT-MBT tasks are sent as console-item objects to the client, if requested in the respective commands.

{"console-item":"<content of console port datagram (base64)>"}

The content of the text message does not follow any format or grammar and is tool dependent.

15.10 Debugging

The client/server architecture and the RTT-MBT tools both do not support interactive debugging but error reporting is implemented as well as functionality to retrieve debugging information.

15.10.1 Error Messages

If problems occur during the execution of commands, the server sends exception objects containing error messages to the client.

{"exception":
{"job":"<command name>",
"job-id" : <job identification number>,
"problems":["<error msg 1>",...,<error msg n>]}}

15.10.2 Debug Information

If debug information should be retrieved by the server and sent to the client, a debug-information object has to be sent to the server before the command object on the same TCP/IP connection. The parameter verbose specifies whether the respective RTT-MBT tool for the actual command is set into verbose mode. The parameter imr-graph specifies whether a PDF file of the abstract syntax tree of the model should be generated or not.
The debug information is sent to the client after the command result object on the same TCP/IP connection. The `debug-result` object contains the base64 encoded content of a zip archive with files that are needed for debugging. The content of the zip archive can be specified in a configuration file for the RTT-MBT server. If no configuration file is provided, a default is used.

```
{"debug-result":
{"debug-information.zip":<base64 encoded debug info zip file>}}
```

### 15.11 Compression

The RTT-MBT server supports gzip compression of every base64 encoded content in the JSON objects. A `compression-settings` object has to be sent after the `client-information` object and before the first object that contains base64 content. A default for compression can be specified as a command line option to the server. The value of an attribute of a JSON object is compressed before the compressed content is base64 encoded. If compression is enabled, the server expects the content from clients to be compressed before base64 encoded.

```
{"compression-settings":
{"compress-base64-content":"true"|"false",
 "compression-method":"gzip"}}
```
Chapter 16

Interaction with Artisan Studio

Artisan Studio is the modelling tool that is used in the COMPASS SoS context to generate SoS models, as well as test models for RTT-MBT test procedure generation. The COMPASS work flow defines two possible export formats from an Artisan Studio SysML model that can be used as input to the COMPASS tool suite: CML and XMI\textsuperscript{1}. The CML export is a representation of the SysML model using the CML. While CML and the CML representation of SysML models is part of the COMPASS development, XMI is a standardised export format for UML/SysML models that has been defined by the Object Management Group (OMG) and XMI representations of UML and SysML models are currently used as a model exchange format between modelling tools.

16.1 XMI Export

Artisan Studio provides a XMI export that allows to generate an XMI representation of the complete model. If only parts of the model are used as a test model, the information where to find the test model within the complete SysML model has to be provided to the RTT-MBT parser. The RTT-MBT parser generates an abstract syntax tree (AST) from the XMI representation that is used as input to the RTT-MBT tools. The AST representation is described in detail in \textsuperscript{7}.

The test model can be a separate model or can be included as a separate package in the SoS development model. If no code generation is used to

\textsuperscript{1}Described in detail in \textsuperscript{7.1}
derive the SoS implementation or parts of the SoS implementation from the development model, the model or parts of it can be used for test procedure generation, as well. The supported SysML subset is listed in [16.1]. If parts of the development model should be used as input for the RTT-MBT test procedure generation process, these parts have to be compliant with requirements for test models. In addition to the requirement that only the supported SysML subset is used, there are some modelling requirements that have to be taken into account. These are described in section and [7.2].

### 16.2 CML Export

The CML export functionality of Artisan Studio is developed during the COMPASS project. This CML representation of the SysML model is parsed by the CML parser of the COMPASS CML tool suite to generate an abstract syntax tree. Similar to the AST from the XMI export, this AST is used as an input to the CML tools. The current implementation of RTT-MBT tools and CML tools do not contain functionality to use the CML-AST as input to the RTT-MBT tools or the XMI-AST as input to the CML tools.

The Eclipse integration of the RTT-MBT tools into the COMPASS Eclipse framework can be used to access RTT-MBT functionality and could be used in the future to exchange information between the XMI-AST and the CML-AST. Plans and ideas for this data exchange are described in the following chapter [17].
Chapter 17

Interaction with CML Tool Components

The CML tools of the COMPASS tool suite address the verification tasks proof obligation generation, theorem proving, model checking and refinement checking. The value of finding problems and failures in the design phase already or proving correctness of a design is widely accepted, as well as the fact that tool support is needed for these tasks for complex system, such as SoS.

The RTT-Mbt tools can be used to generate and execute test procedures and collect information about the test results. Though test procedures are normally executed on an existing SUT implementation, the test procedure generation is performed on a model of the SUT that can be developed in an early design phase. To support early test suite validation, RTT-Mbt allows to generate SUT simulations from the model, so that model-in-the-loop tests can be performed in absence of “real” SUT software or integrated HW/SW systems.

In addition to generating test procedures, RTT-Mbt tools also provide functionality to run checks a model, e.g. check for potential live-locks in state machines or check if LTL formula can be satisfied for a test model (see Chapter 13).

All test and verification tools are integrated in the Eclipse base COMPASS tool IDE. This section describes concepts and future plans about how the tools can exchange results and use the output from other tools in the COMPASS tool suite as additional input.
17.1 AST Transformation

The RTT-MBT tools currently only accept XMI model representations as input and use this to generate the XMI-AST. It is also possible to define a transformation from the CML-AST or at least parts thereof into the XMI-AST so that the RTT-MBT tools could also be used with models that exist only as CML representations. This transformation could be defined on AST level and would avoid the implementation of a second CML parser.

17.2 Coverage Information

Test procedures generated from a test model by RTT-MBT tools can automatically trace requirements coverage and verdict, if the requirements tracing is modelled correctly in the test model. This information can be transferred as additional information to requirements in the CML-AST that is used by the CML tools or to the COMPASS tool IDE. Though the requirements coverage or verdict information is less relevant for model checking or theorem proving, it is of great value for reports about the SoS.

17.3 Bounded Model Checking

As described in Chapter 13, RTT-MBT supports bounded model checking against SafetyLTL formulas. The BMC checker could accept checking requests from other verification tools and return witness traces to them if solutions for the property exist.

Proof obligation generated by the CML tools could be passed to RTT-MBT instead of trying to prove it using the theorem prover component. If the proof obligation could be transformed into goals for the test generator, a test procedure could be generated to demonstrate that the goal can or cannot be reached.

For local livelock checking of (small sets of cooperating) state machines RTT-MBT provides a checker optimised for this task.
17.4 Exploiting Symbolic Test Cases Information

Symbolic test cases generated by RTT-MBT are SafetyLTL formulas, and, in order to create concrete test data, the tool creates witnesses for these formulas. These existentially quantified facts ("there exists a computation fulfilling \( \phi \)") could be passed on to other verification tools that might use them, for example, to falsify assumptions.
Part IV

RT-Tester Tool Qualification
Chapter 18

Introduction to Tool Qualification

The application of test automation tools in a safety-critical context requires so-called tool qualification according to the applicable standards. The objective of this qualification is to justify that verification steps automated by the tool will not lead to faulty systems under test to be accepted as fit for purpose.

A major part of the work presented in this part has been published in [BPS12]. The first author’s contribution has been developed during the course of the project “Verifikation von Systemen synchroner Softwarekomponenten” (Ver-SyKo) funded by the German ministry for education and research (BMBF). The second and third authors’ work has been elaborated within COMPASS.

18.1 Model-based Testing and Associated Tool Support

In model-based testing, a test model is used to define the expected behavior of the system-under-test (SUT). From this formal specification of the desired system behavior, test cases are generated, which are then executed against the SUT. The generation of test cases is frequently based on techniques such

\footnote{It is important to note that correctness of model-based test generators relies on the consistency and completeness of test models, since test cases are derived directly from the}
as abstract interpretation \cite{CC77} and SMT solving \cite{KS08}, which exercise the semantic structure of the model to automatically calculate these test cases; such techniques are also implemented in our tool-suite RT-Tester Model-Based Test Generator (RTT-MBT) \cite{PVL11a}. The resulting test cases are then specified as sequences of input data — including timing constraints — that stimulate computations of the SUT conforming to the test case specifications. In addition to the input stimulations, RTT-MBT automatically generates test oracles that run concurrently with the SUT, checking the responses of the SUT against the test model on-the-fly. In combination, the stimulation component and the test oracles form a test procedure which is compiled and executed in a test execution environment (TE). The test execution environment then generates a so-called execution log, which contains the data observed and recorded during test case execution.

18.2 Model-based Testing in Industry and Tool Qualification

The success of model-based testing in industry has been stimulated by the success of model-driven software development in general. Indeed, compared to conventional approaches, model-based testing has proven to increase both quality and efficiency of test campaigns \cite{LP10}, which may explain the industrial interest in model-based methods, especially from domains such as automotive, avionics, and railway industry. However, all tools that automate process steps in the development and verification of safety-critical systems (e.g., code generators, compilers, model checkers, test automation tools) need to be qualified since they automate a life cycle activity so that a manual inspection of its outcome (e.g., generated source code, object code, verification or test results) is rendered superfluous. In this situation it has to be ensured that the tool performing this automation cannot inject errors into the artefacts produced; otherwise, this would induce the risk of a faulty system component to be accepted as fit for purpose.

The ISO 26262 \cite{ISO09} standard currently implemented in the automotive domain presents guidelines for the development of safety-related systems in road vehicles. This standard is an exemplar of a regulation prescribing required properties for development and verification automation tools. The standard \cite[Section 11.4]{ISO09} itself expresses the aim of tool qualification (TQ) as follows:

models. It is thus assumed that the test models have undergone review.
“The objective of the qualification of software tools is to provide evidence of software tool suitability for use when developing a safety-related item or element, such that confidence can be achieved in the correct execution of activities and tasks required by ISO 26262.”

The key steps of providing the required evidence are defined in the standard. First of all, qualifying a tool for a development process necessitates to determine the tool impact (“what are the worst possible effects of tool malfunctions on the SUT?”) and its error detection capabilities (“will tool malfunctions be revealed?”). These two factors are combined to form an overall tool confidence level. Of course, the more critical the developed system component, the stricter the requirements imposed onto the tools automating development or V&V activities. Yet, an interesting aspect of ISO 26262 [ISO09, Section 11.4.4.1] is that tools with maximal tool error detection capabilities do not require qualification measures at all, as long as error detection is perfectly reliable. Hence, if a tool is capable of detecting its own malfunctioning, the entire tool qualification process for ISO 26262 can be simplified in a significant way.

18.3 Tool Qualification According to ISO 26262

Malfunction of a test-case and test procedure generator such as RTT-MBT introduces two hazards which may result in a situation where a requirement allocated to a safety-related item is violated, due to malfunction of this item:

- **Hazard 1: undetected SUT failures.** A deviation of the observed SUT behavior from its expected behaviour specified in the test model may potentially remain undetected if the generator creates erroneous test oracles failing to detect this deviation during test executions.

- **Hazard 2: undetected coverage failures.** The test execution fails to meet the pre-conditions which are necessary in order to cover a given test case, but the test oracles indicate TEST PASSED because the observed execution is consistent with the model. If this situation remains undetected, it may be assumed that the SUT performs correctly.

---

2That is, a software or embedded HW/SW control system tested by means of procedures generated by the tool.
with respect to the specified test case, while in fact the test procedure tested “something else”.

The qualification goal required by ISO 26262 states that any possible hazard introduced by the tool will eventually be detected [ISO09, Section 11.4.3.2]. The identification of components of model-based test generators relevant for qualification, as well as trustworthy, yet lightweight methods for satisfying this objective are topics of this part. Of course, formal verification of RTT-Mbt as a whole to prove the absence of defects is an unrealistic mission, as the state-of-the-art represents verifying functional correctness of systems that involve approximately 10,000 lines of C code [KAE+10], and RTT-Mbt consists of approx. 250,000 lines of C/C++ code. To qualify RTT-Mbt for use in the development of software of high quality assurance levels (according to ISO 26262 and other standards such as RTCA DO-178C or CENELEC EN50128-2011), it is thus necessary to combine formal verification with testing and effective tool error detection. In the following, we discuss the verification strategies we applied to RTT-Mbt and the tool error detection mechanism.

The key idea of our approach to tool qualification is simple: rather than attempting a priori verification for the test generator by proving conformance of generated test procedures with the model, we focus on the a posteriori error detection capabilities of RTT-Mbt, regarding the consistency of the test execution log with the model. This is performed by replaying the execution log on the test model. The key functionality for replay of test execution is defined as follows:

- A simulation of the test model is executed that uses exactly the input data to the SUT that was used during test procedure execution.

- The respective simulation determines the expected SUT outputs as calculated from the test model. These outputs are compared to the outputs observed during test procedure execution and documented in the test execution log. Any deviations are recorded in the replay verification results.

- The actual model and test coverage achieved by the simulation is recorded in the replay verification results, too.

Using this strategy, the impact of errors in the test generator is localised. In essence, it suffices to show that replay detects any deviation of the concrete test execution from the test model. Establishing this correctness argument is strictly easier than proving correctness of the entire test generation functionality. In principle, replay could be performed on-the-fly, concurrently
to the test execution. Yet, we describe it as an *a posteriori* activity, to be performed offline after the execution has been completed, since hard real-time test engines running HW/SW or system integration tests often do not have sufficient spare computing power to cope with additional model executions for replay purposes.

### 18.4 Contributions

In summary, this part presents the following contributions:

- We present an analysis relating properties of model-based test generators to requirements for tool qualification imposed by ISO 26262.
- We identify classes of hazards introduced by test generators and provide an analysis of parts of a test generator that are relevant to qualification.
- We introduce a lightweight framework for replaying concrete test executions with the aim of identifying erroneous test case executions.
- We show how the different verification activities, consisting of design analyses, formal verification, structural testing and tool error detection are combined to form a convincing justification of the tool qualification, to be presented to customers and certification authorities.
- We compare the TQ-related differences in the relevant standards from the avionic, railway and automotive domains.

### 18.5 Outline of Part IV

The exposition of this part is laid out as follows. First, Chapter 19 presents an impact analysis that connects the test generator to the demands imposed by the ISO 26262. Then, Chapter 20 identifies those parts of a test generator that are relevant for tool qualification and discusses the details of the verification strategies applied. Following, Chapter 21 studies properties of the replay and constructs a correctness argument from the architecture. Chapter 22 discusses TQ-related differences between ISO 26262, RTCA DO-178C and CENELEC EN50128-2011. This is relevant for test tools applied in a SoS context where standards from different domains apply, and tools have to be qualified across domains. We apply the results of this comparison in Chapter 23 where the TQ requirements for MBT tools to be used for SoS
testing in the automotive, avionic and railway domains are identified. Finally, the part concludes with a survey of related work in Chapter 24 and a final discussion in Chapter 25.
Chapter 19

Tool Classification According to ISO 26262

To assess the qualification requirements for a model-based test-case generator, it is necessary to analyse hazards potentially created by the tool, as well as the impact of these hazards. The second important aspect is the tool error detection capability with respect to its own malfunction. Based on this analysis, particular requirements are imposed onto the tool, which are discussed in the remainder of this section.

19.1 Impact Analysis

ISO 26262 defines two different tool impact (TI) levels for software tools of any kind: TI0 is applicable if and only if malfunctioning of the tool can under no circumstances introduce a hazard; otherwise, TI1 shall be chosen [ISO09, Section 11.4.3.2].

Assuming correctness and completeness of the test model,\(^1\) potential malfunction of a model-based test-case generator introduces the hazards 1 (undetected SUT failures) and 2 (undetected coverage failures) introduced in Section 18.3.

\(^1\)Since models represent abstractions of a real system, correctness and completeness is usually defined according to some conformance relation between model and SUT. In our case – since the SUT never blocks inputs – conformance means that for any given timed trace of inputs (1) the SUT produces the same observable outputs as the model, modulo tolerances regarding timing and floating point values, and (2) that the ordering of inputs and outputs, when restricted to a sequential sub-component of the model, is the same for model and SUT (partial ordering of observable I/Os).
These hazards clearly imply tool impact level \textbf{TI1} for model-based test-case generators.

\section{19.2 Tool Error Detection Capabilities}

The probability of preventing or detecting an erroneous tool output is expressed by tool error detection classes \textit{TD} \cite{ISO2009} with range \textit{TD}$_1$, \ldots, \textit{TD}$_4$. If there is high confidence in the ability of a tool to detect its own malfunctioning, then \textit{TD}$_1$ is the appropriate classification. Lower classes such as \textit{TD}$_3$ or \textit{TD}$_4$ are applicable if there is low confidence in the tool’s error detection facilities, or if no such mechanism exists. To achieve \textit{TD}$_1$, we specify one tool-external measure — that is, a guideline to be respected by the test engineers applying the tool — and three tool-internal measures, that is, measures implemented in software.

\textbf{External Measure.} Every test execution of test procedures generated by RTT-MBT shall be replayed.

Independent of the specific functionality of RTT-MBT, this external measure is mandatory to enforce anyway, since every testing activity for safety-critical systems requires a verification of the test results. The replay function as introduced in Section 18 can be considered as a tool-supported review of this kind, because it verifies whether the test execution observed complies with the model and whether the intended test cases have really been executed. To support the external measure above, or, equivalently, to achieve \textit{TD}$_1$, we implement three tool-internal measures with the following objectives:

\textbf{Internal Measure \#1.} Every change of input data to the SUT is correctly captured by logging commands in the test stimulator of the test procedure.

\textbf{Internal Measure \#2.} Every change of SUT output data is correctly captured by logging commands in the test oracles of the generated test procedure.

\textbf{Internal Measure \#3.} The replay mechanism detects (1) every deviation of the SUT behavior observed during test execution from the SUT behavior expected according to the test model, and (2) every deviation of the test cases actually covered during the test execution from the test cases planned to be covered according to the test generator.
The effectiveness of internal measures 1 — 3 clearly implies the desired TD1 capability, as long as test engineers comply with the external measure above.

For the tool-internal measures to be effective, two prerequisites of the overall test system need to be satisfied: (1) All logging commands in the test procedures are correctly executed and recorded in the execution log, and (2) the test model is complete and correct. Clearly, the first prerequisite is delegated to the test execution environment, a component which is independent of the functionality discussed in this report and, thus, must be qualified on its own. Satisfying the second prerequisite is a duty of test engineers that use a model-based test generator.

19.3 Tool Confidence Level

Pairing the tool impact TI with the appropriate tool error detection capability TD yields the associated tool confidence level TCL [ISO09, Section 11.4.3.2]. It is remarkable that for a tool with impact level TII and error detection class TD1 (such as Rtt-Mbt), no qualification whatsoever is required according to [ISO09, Section 11.4.4.1]. Otherwise, tool qualification measures have to be adopted according to the automotive safety integrity level (ASIL) of the system under test. This situation is depicted in Fig. 19.1. High confidence in the error detection capabilities of a tool thus eases the tool qualification significantly, which motivates the desire for trustworthy, lightweight error detection integrated into the tool.
Figure 19.1: Relation between tool impact (TI), tool error detection class (TD), tool confidence level (TCL), and automotive safety integrity level (ASIL) of the SUT.
Chapter 20

Verification for Qualification

The overall architecture of RTT-MBT is depicted in Fig. 20.1. A parser component translates an input model written in UML or SysML, which is given as an XML export of some modeling tool, into an intermediate model representation (IMR, the abstract syntax tree of the model) that is used by both the test-case generation and the replay facilities. The test-case generator uses techniques such as SMT solving, abstract interpretation and code generation in order to generate test inputs and the corresponding test
Table 20.1: Functionality in RTT-MBT affected by tool-internal measures for error detection

<table>
<thead>
<tr>
<th>Measure</th>
<th>Affected Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>#1</td>
<td>The method that creates the log commands associated with SUT inputs.</td>
</tr>
<tr>
<td>#2</td>
<td>The method that creates a conditional logic operation that shall be executed whenever an SUT output observed is changed. The method that creates the test oracles and inserts a log operation to be executed whenever an SUT output observed by this test oracle is changed. The method that stores the conditional log operations. The method that creates the log operation in the syntax of the test oracle.</td>
</tr>
<tr>
<td>#3</td>
<td>The method that parses an execution log. The interaction between the replay log and the simulator. The methods that simulate an input from the replay log. The method comparing the observed SUT outputs with the expected SUT outputs calculated by the simulator. The methods that manage the internal memory during simulation. The methods that determine whether the test cases covered during replay are identical to those observed during a concrete execution.</td>
</tr>
</tbody>
</table>

procedures. In contrast to this, the sole purpose of the replay mechanism is to simulate an execution of a concrete execution log on the test model. Test-case generation and replay are thus independent components that have also been developed with independence. The test generator, however, is responsible for the generation of log commands in test procedures, and the replay mechanism depends on their correct generation (recall the tool-internal measures discussed in Section 19.2). Therefore this part of the generator (it is a sub-component of the test procedure generator) receives increased attention during verification.

20.1 Identification of Relevant Components

The parser is responsible for translating an input model into an IMR. Therefore it must be qualified as errors in this component may mask failures of the test
generator and the replay. To identify classes and methods reachable from
the parser, we use data-flow analysis and code inspection. Apart from the
parser itself, the relevant classes most notably include the IMR used within
Rtt-Mbt.

As argued in Section 19.2, a replay that implements the tool-internal measures
eliminates the need to qualify test generation. This strategy entails that the
replay needs to be qualified, as tool error detection is delegated to the replay
component. It is noteworthy that test generation is much more complex
than replay. Complementing test generators with replay mechanisms thus
reduces the workload for qualification significantly: the existence of tool error
detection turns test generation — with the exception of the log command
generator — into a component whose outputs need not be verified. As before,
we apply data-flow analysis to identify classes and methods that are involved
during replay. The result of this analysis includes, most notably, the IMR,
the memory model storing states during model simulation, the simulator, and
the parser for the test execution log.

20.2 Identification of Tool-internal Measures

Additionally, we performed a design review to identify those parts of Rtt-Mbt
implementing the tool-internal measures 1—3 for error detection. The results
of this analysis are given in Table 20.1. These parts must be implemented
correctly under any circumstances. Special attention should thus be paid
during the verification of the methods highlighted in Table 20.1.

20.3 Verification Activities

To verify the correctness of each involved component, we apply different
verification activities. The main verification activity is (software integration)
testing, though this process is augmented with formal verification for the
most critical software parts. The combination of methods applied conforms
to the requirements of RTCA DO-178B for developing software of the highest
criticality (Level A): (1) The development process for the replay component
has been controlled according to [WG-92, Table A-1,A-2]. (2) High-level
and low-level requirements specified for the replay mechanism have been
verified with independence according to the approach defined in [WG-92,
Table A-3,A-4]. (3) The source code has been inspected, formally verified
and tested according to [WG-92] Table A-5, A-6, A-7. (4) The configuration management and software quality assurance processes have been set up for the whole RTT-MBT product in a way conforming to [WG-92] Table A-8,A-9, respectively.

**Requirements-based testing.** For each high-level and low-level requirement of the test generator, we provide normal behavior tests as well as robustness tests that investigate the stability of RTT-MBT.

**Structural testing.** We provide a collection of test cases that achieve MC/DC coverage for the reachable parts of RTT-MBT.

**Formal Verification.** For certain critical components, we perform additional formal verification and documentation of the verification results. An example of functionality for which proofs are provided are the logging facilities. Formal verification is performed for small isolated functions specified by pre- and post-conditions. The verification is performed manually using Hoare Logic. The small size of the verified functions indicated that manual proofs were acceptable and could be checked independently by a verification specialist.

A summary of the verification activities for each component is given in Table 20.2. We define the following general criteria for the testing process of the involved components, before discussing the details for the strategy applied to the parser.

**Pass/fail criteria.** The test passes if the expected results are produced

<table>
<thead>
<tr>
<th>Component</th>
<th>Requirements</th>
<th>Structural testing</th>
<th>Formal Verification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Measure #1</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Measure #2</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Measure #3</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Parser</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>IMR</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Replay</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Generation</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>
without deviation; otherwise, it fails.

**Test end criteria.** The test ends when all test cases have passed and the test suite resulted in a 100% MC/DC coverage for the involved methods.

**General test strategy.** Each method shall be tested as follows:

- The test cases for verifying the methods are elaborated.
- A test model is selected which is suitable for the test cases under consideration. Test cases shall cover both normal behavior and robustness.
- A replay file is selected which is based on a test execution whose test procedure was generated with the selected model and which is suitable for the test cases under consideration.
- The replay function is activated with the model and replay file as inputs.
- The replay results consist of the pass/fail results achieved during the replay, and the list of test cases covered during the replay.
- The replay results are verified with respect to the expected results. It is checked that the replay result is pass if and only if the replay file corresponds to a correct model computation. Evaluating the replay file against the model, it is checked against the model that the list of covered test cases produced during replay is correct and complete.
- The MC/DC coverage achieved during the execution of the test suite is checked whether it results in 100% for the methods identified above.
- If some code portions could not be covered by the test suite designed according to the guidelines listed here, it is admissible to perform unit tests on the methods not completely covered so far. The unit tests shall check the expected results by means of post conditions. The associated unit test cases specify the data needed to cover the missing code portions as test condition, and the post condition as expected result.
20.3.1 Test Strategy for the Parser Component.

The IMR of a model is the basis for all further functionality of RTT-MBT. Errors in the IMR can therefore mask failures in the test data generation as well as in the replay. Thus the parser component setting up the IMR has to be qualified. The IMR is a representation of the abstract syntax tree (AST) of the model restricted to those UML/SysML elements which are supported by RTT-MBT. An example of a simple state machine, which is part of a test model, is shown in Fig. 20.2. The corresponding IMR is shown in Fig. 20.3. In addition to the IMR, a model of the internal memory representation is generated, which is used during simulations to store the values of variables and control states. Both the AST and the memory model are used as inputs to the replay and must thus be qualified.

For requirements-based tests of the parser, models serve as inputs that have been designed to specifically test one or more syntactic UML/SysML feature supported by the parser. Their respective XML representations are the test inputs to the parser, which generates both, the IMR and the memory model, from the test model. The IMR is then verified to be a valid representation of the model. For each test model, we handcraft an AST corresponding to the expected IMR. The result generated by the parser is then checked against this expected AST.

The memory model is an algorithmically simple component. Qualification amounts to verifying that for each control state and each variable, an appropriate entry in the internal memory is initialized. Whenever a value is stored in the memory, its internal state shall correctly reflect the update. It also needs to be ensured that support for data types is exhaustive. These properties are checked using a combination of requirements-based integration tests and unit tests which, e. g., systematically probe all data types. The
inputs for the requirements-based test procedures form equivalence classes. The structural coverage tests refine these equivalence classes so that MC/DC coverage is eventually achieved.

As explained above, the verification strategy for the parser merely analyzes the correct parsing and transformation of a pre-defined class of language patterns. This approach dovetails with strategies applied to non-optimizing compilers [FFFL+11, Section 2.1]. There, the strategy is to verify the correct translation of each supported symbol in a high-level programming language into the corresponding assembly fragment. Since RTT-MBT does not optimize the AST, such a direct mapping from expected inputs to outputs can also be established for the model parser.

Figure 20.3: IMR of the state machine in Fig. 20.2
Chapter 21

Error Detection Using Replay

Objectives. In this section we elaborate a formal argument to show that replay — if correctly implemented — enforces the tool error detection class TD1 as required. Let us briefly recall the scenarios that may occur once a generated test case is executed: (1) If test execution fails, manual investigation is required to justify the deviations, identify an erroneous test case, fix the SUT, or refine the test model. (2) If test execution passes, this may be due to one of the following reasons: (a) The test cases were generated correctly from the test model and the SUT conforms these test cases, or (b) some test cases were incorrectly generated[1] from the test model, but the SUT behavior still conforms to these faulty cases. Tool error detection thus needs to classify only test case executions that pass. In the following, we build towards a correctness argument for the replay.

Prerequisites. The SUT S communicates with the test environment through finite sets of input signals $\mathcal{I} = \{i_1, \ldots, i_m\}$ and output signals $\mathcal{O} = \{o_1, \ldots, o_n\}$ such that $\mathcal{I} \cap \mathcal{O} = \emptyset$. We denote the set of overall signals by $\mathcal{S} = \mathcal{I} \cup \mathcal{O}$. Further, each signal $s \in \mathcal{S}$ is assigned a type drawn from a finite class $\mathbb{T} = \{\mathbb{N}, \mathbb{B}, \mathbb{Q}, \ldots\}$ of types using a map $\sigma: \mathcal{S} \to \mathbb{T}$. An input to the SUT is then given as a triple $\langle t, i, v \rangle$ where $t \in \mathbb{R}$ is a time-stamp, $i \in \mathcal{I}$ is an input signal and $v \in \sigma(i)$ is an assignment to $i$. Likewise, an output from the SUT to the test environment is a triple $\langle t, o, v \rangle$ with $t \in \mathbb{R}$, $o \in \mathcal{O}$, and $v \in \sigma(o)$.

[1]This means that either the test data is inappropriate for the test objective, or the check of expected results is faulty.
**Model Semantics.** The desired behavior of the SUT is specified by means of a (deterministic) test model $M$. The model $M$ is syntactically reproduced by a parser that has been qualified. Syntactic correctness of $M$ can thus be assumed. Semantically, $M$ can be interpreted as the (possibly infinite) set of (infinite) computation paths it defines, which we denote by $[M]$. Since $M$ is deterministic, a path $\pi \in [M]$ is uniquely determined through its observable input-output behavior, e.g., partial (timed) assignments of the above form to the signals in $S$.

**Test Generation.** Formally, a test generator is a function that computes a finite set of finite traces from $M$, which we denote $[M]_{TC}$. If correct, each $\pi_{TC} \in [M]_{TC}$ is the finite prefix of a computation path $\pi$ through the test model (the prefix relation is denoted by $\pi_{TC} \prec \pi$). Since the test generator is not qualified, and is thus an untrusted component, we have to assume that such a corresponding path does not necessarily exist. We define:

**Definition 1** Let $\pi_{TC} \in [M]_{TC}$. The predicate $\text{correct}$ on $\pi_{TC}$ is defined as:

$$\text{correct}(\pi_{TC}) \iff \exists \pi \in [M] : (\pi_{TC} \prec \pi)$$

**Test Execution.** Given $\pi_{TC} \in [M]_{TC}$, the test procedure that executes $\pi_{TC}$ generates an execution log that can be interpreted as a finite trace $\pi_{exec}$. As before, we interpret the semantics $[S]$ of an SUT $S$ as the set of its feasible execution traces; clearly, $\pi_{exec} \in [S]$. Observe that $\pi_{exec}$ is not required to be identical to $\pi_{TC}$: timings and floating point values may deviate within specified limits, and only a partial ordering of I/Os has to be observed as explained in Section 19.1. However, if $\pi_{exec}$ conforms to $\pi_{TC}$ the test passes, which we denote by $\text{pass}_{\pi_{TC}}(\pi_{exec})$. Since the execution log is compared to a trace from an untrusted generator, we cannot infer correctness of the test execution with respect to $M$.

**Proposition 1** Let $\pi_{TC} \in [M]_{TC}$ and $\pi_{exec} \in [S]$. Then

$$(\text{correct}(\pi_{TC}) \land \text{pass}_{\pi_{TC}}(\pi_{exec})) \Rightarrow (\exists \pi \in [M] : \text{pass}_{\pi}(\pi_{exec}))$$

**Replay.** Formally, the replay mechanism can be interpreted as a predicate replay:
Definition 2 Let $\pi_{\text{exec}} \in [S]$ denote an execution of $\pi_{\text{TC}} \in [M]_{\text{TC}}$ such that $\text{pass}_{\pi_{\text{TC}}} (\pi_{\text{exec}})$. The predicate $\text{replay} : [S] \to \mathbb{B}$ is defined as:

$$\text{replay}(\pi_{\text{exec}}) \iff \exists \pi \in [M] : (\pi_{\text{TC}} \prec \pi) \land \text{pass}_{\pi_{\text{TC}}} (\pi_{\text{exec}})$$

Since correctness of the implementation of the replay mechanism is ensured, we safely assume that $\text{replay}(\pi_{\text{exec}}) = \text{true}$ iff $\pi_{\text{exec}}$ is the prefix of a path $\pi \in [M]$ such that $\text{pass}_{\pi} (\pi_{\text{exec}})$. We thus obtain the correctness argument:

Proposition 2 Let $\pi_{\text{exec}} \in [S]$ denote an execution of $\pi_{\text{TC}} \in [M]_{\text{TC}}$ such that $\text{pass}_{\pi_{\text{TC}}} (\pi_{\text{exec}})$. Then, $\text{correct}(\pi_{\text{TC}}) \iff \text{replay}(\pi_{\text{exec}})$.  

In consequence, given a generated test $\pi_{\text{TC}} \in [M]_{\text{TC}}$ and an execution $\pi_{\text{exec}} \in [S]$ that passes, $\pi_{\text{TC}}$ is the finite prefix of a path $\pi \in [M]$ iff $\text{replay}(\pi_{\text{exec}})$ as desired, which ultimately provides a proof of correctness of the mechanism.
Chapter 22

TQ-Related Requirements – Comparison Between ISO 26262, RTCA DO-178C and EN50128:2011

22.1 Qualification of SoS V&V Tools

When developing SoS with constituent systems from different domains, any tool used across these domains has to comply with the qualification guidelines from all domains where qualification is required. This chapter exemplifies such cross-domain qualification requirements by analysing the applicable standards from the automotive, avionic and railway domains.

- ISO26262 [ISO09] is the standard applicable for the automotive domain, our qualification strategy described in the previous chapters focused on this standard.

- RTCA DO-178B [WG-92] for the avionic domain was one of the first standards to explicitly address tool qualification. It is now replaced by the updated standard RTCA DO-178C [WG-11] devoting a whole companion standard [RTC11] to tool qualification.

- Since 2011 the railway domain has also specified its own TQ requirements in the CENELEC standard EN50128:2011. [CEN11]. Observe that in contrast to the previous standards, EN50128:2011 uses the term tool validation instead of tool qualification.
22.2 Characteristic TQ-Related Concepts

**TQ-applicability – qualification only for tools replacing manual processes.** All standards explicitly state that TQ is only required if tool application is intended to replace a manual process by an automated one. TQ is only required to demonstrate that the artefacts produced automatically by the tools can be relied upon *without manual inspection*. In any situation where tool output is verified with the same effort as would have been applied in the case of manual creation of the equivalent output, TQ is not necessary.

If, for example, a model-based code generator is applied, but the generated code is always tested against the model with the same effort as would have been required by the standards in case of manual code development, the generator does not have to be qualified. If, however, only the model is verified and the generated code taken to be “automatically” consistent with the model, then qualification of the generator is necessary.

**Project/product-specific tool qualification.** All three standards emphasise that tool qualification cannot be unconditionally granted for any development or verification tool, but has to be performed with respect to a specific project or product to be developed (see [ISO09, 11.1], [WG-92, 12.2.1], [RTC11, p. 5, and 11.2.1] and [CEN11, 6.7.4.1]). Practically speaking, certain tool components may only be universally qualified if it can be justified that these components’ behaviour does not depend on a specific project or target system to be developed, and if the component has already been qualified for the highest criticality level. All other components have to be re-qualified for each development and verification campaign. For test automation tools this implies that the interfaces between test tool and SUT have to be specifically qualified, because correctly calculated test data may be passed along the wrong interfaces to the SUT. Conversely, erroneous SUT outputs may be passed along a wrong interface where the data appears to be correct. Moreover, it has to be verified that the interface-specific refinements and abstractions performed by the test automation tool (e.g., transforming abstract values used on model level to concrete communication telegrams and vice versa) are correct.

**Tool impact assessment.** All three standards require that the qualification effort to be invested shall depend on the impact that tool malfunctions could have on the target system under consideration. The automotive standard differs, however, in one crucial aspect from the others.
- ISO26262 classifies the criticality of a tool alone on the basis whether erroneous tool behaviour may result in erroneous (safety-related) target system behaviour. If this can never be the case, the tool is classified to have impact level TI0. Otherwise it has impact level TI1 [ISO09, Section 11.4.3.2].

- RTCA DO-178C further distinguishes between development tools whose outputs become part of the airborne software (Criteria 1) and verification tools (including test automation tools) whose malfunction could only lead to an error in the target system remaining undetected (Criteria 2). Tools without impact on the target system are classified as Criteria 3 [WG-11, Section 12.2.2]).

- EN50128:2011 classified the tool impact in conformance with RTCA DO-178C and uses the terms
  - T1: tool does not produce outputs directly or indirectly influencing the executable code of the target system.
  - T2: tool influences the executable code of the target system only in an indirect way, because it may produce and/or overlook failures in artefacts that are not part of the target system (such artefacts are, for example, design documents and test procedures or test results).
  - T3: the tool produces artefacts that are directly deployed on the target system (e.g., executable code or configuration tables).

Table 22.1: Comparison between tool impact classification in ISO26262, RTCA DO-178C and EN50128:2011.

<table>
<thead>
<tr>
<th>Standard</th>
<th>Tool Impact</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTCA DO-178C</td>
<td>Criteria 1</td>
</tr>
<tr>
<td></td>
<td>Criteria 2</td>
</tr>
<tr>
<td></td>
<td>Criteria 3</td>
</tr>
<tr>
<td>EN50128:2011</td>
<td>T3</td>
</tr>
<tr>
<td></td>
<td>T2</td>
</tr>
<tr>
<td></td>
<td>T1</td>
</tr>
<tr>
<td>ISO26262</td>
<td>TI1</td>
</tr>
<tr>
<td></td>
<td>TI1</td>
</tr>
<tr>
<td></td>
<td>TI0</td>
</tr>
</tbody>
</table>

**Tool error detection capabilities.** Both the automotive and the railway standards point out that a tool’s functional correctness need not be verified in detail if the tool is equipped with a reliable error detection mechanism [ISO09, Section 11.4.3.2], [CEN11 6.7.4.4]. To this end, ISO26262 introduces error
detection levels TD1 (high degree of confidence that errors in tool outputs will be detected) to TD4 (no trustworthy error detection capabilities available). It is remarkable that this possibility of error detection is not addressed in the TQ requirements from the avionic domain.

Tool confidence levels. ISO26262 is unique among the standards discussed here in that it derives a tool confidence level from its impact and error detection levels: tools with no impact on the target (TI0) have highest confidence level TCL1, and TI1-tools without error detection capabilities (TD4) have lowest confidence level TCL4 [ISO09, Section 11.4.3.4]. The confidence level is an a priori classification of the tool which helps to determine the qualification effort required: TCL1-tools need no qualification at all, whereas TCL4-tools cannot be used without additional qualification measures [WG-11, Table 12-1].

Qualification levels. Both the automotive and the avionic standards relate the qualification effort to the criticality of the target system for whose development or V&V the tool is planned to be applied. The avionic standard introduces the term tool qualification level for the qualification effort required, with a level range from TQL1 (highest effort) to TQL5 (lowest effort). While ISO26262 determines this effort on the basis of target system criticality and tool confidence level [ISO09, Section 11.4.4.2], RTCA DO-178C determines it on the basis of criticality and impact criteria (since the concept of tool error detection is not considered in RTCA DO-178C at all). The relation between tool and target system criticality is not considered in the railway standard: here the qualification effort is determined alone on the basis of the tool impact and its error detection capability [CEN11, 6.7.4.3 - 6.7.5.12].

DO-178C assigns only tool qualification level TQL4 to tools that automate verification and test of software of the highest criticality (i.e., Level A) [WG-11 Table 12-1], because test tools only have an indirect impact on the target system. TQL4 requires the elaboration of operational and functional requirements and their verification, the verification of protection mechanisms, and requirements-based testing [WG-11 Table T-0 – T-7]. Yet, neither tests against the detailed design, nor code coverage of any measure are required.

By way of contrast, ISO26262 assigns tool confidence level TCL1 to tools with impact TI1 and detection capabilities TD1; this is the situation considered here in the previous chapters [ISO09 Section 11.4.3.2]. Since no qualification

\[\text{1For tools creating (parts of) airborne software the stricter levels TQL1,2,3 are assigned.}\]
whatsoever is required for TCL1, no requirements-based testing is needed. It remains to prove, however, that the tool indeed fulfils TD1 (“there is a high degree of confidence that a malfunction or an erroneous output from the software tool will be prevented or detected”, [ISO09, Section 11.4.3.2, b]). It is remarkable that the standard does not elaborate on how error detection capabilities should be validated. From the general qualification requirements [ISO09, Section 11.], however, we conclude that this should be done with the highest possible effort associated with the target system’s criticality. For the highest criticality level (denoted ASIL D) the standard requires alternatively the evaluation of the development process in combination with a comprehensive validation or — this is the procedure applied for R TTC-MB7 in this report — development in compliance with a safety standard, such as RTCA DO-178B [ISO09, p. 23, Table 2, Ex. 3]. This implies, for example, that detailed tests to achieve MC/DC code coverage have to be performed for the replay.

We conclude that the tool qualification requirements of ISO26262 and RTCA DO-178C / DO-330 are complementary in the sense that the former put emphasis on an in-depth verification of the error detection mechanism with highest confidence, while the latter requires comprehensive requirements-based testing.

Regarding the qualification of test automation tools, the requirement of the railway standard EN50129 are considerably lower: the qualification effort is determined by the tool’s impact classification as T2 (tool malfunction may only lead to errors in the target system being overlooked, but can never introduce errors into the target in a direct way) alone. For this class it is only required to justify why the tool is adequate for its intended purpose [CEN11, 6.7.4.1], a functional specification or user manual has to be provided [CEN11, 6.7.4.3] as well as a documentation of possible tool malfunctions and associated error detection capabilities [CEN11, 6.7.4.2]; it has to be ensured that only well-defined tool baselines are used [CEN11, 6.7.4.10], and new tool versions have to be re-analysed with respect to qualification aspects [CEN11, 6.7.4.11].

Configuration management. All three standards require that only well-defined tool configuration baselines may be applied, and these baselines have to be identified [ISO09, 11.4.2.1], [RTC11 10.1.5, Table T-8], [CEN11, 6.7.4.10]. The avionic standard specifies configuration management requirements in a way that is more specific than the other standards. Depending on the tool qualification level, one of two control categories have to be applied for the
tool: category 1 requires configuration management of more artefacts than category 2. These artefacts are listed in [WG-11, Table 7-1], the mapping from tool qualification level to control category is listed in [RTC11, Table T-8]. For example, tool qualification levels 1 — 3 require that the life cycle environment (that is, the HW and SW used to develop the tool under consideration) has to be baselined according to control category 1. This is not required for qualification levels 4 or 5.
Chapter 23

Cross-Domain Tool Qualification
Requirements for Model-Based
SoS Testing

Considering the differences in TQ-related requirements listed in the different standards from the automotive, avionic and railway domains, we can summarise that

- no contradictory TQ requirements have been defined,
- certain common requirements have to be enforced with different strength according to the different standards, and
- some complementary requirements are defined.

Cross-domain TQ has to cover all of these requirements, and each one in its strictest form.

In this chapter we apply the comparison between standards from Chapter 22 for capturing the TQ requirements of model-based testing tools to be used for SoS testing across the automotive, avionic and railway domains.

**Impact assessment.** The identification of TQ requirements listed below is based on the assessment that the impact of an MBT tool is classified as given in Table 23.1.
Table 23.1: Tool impact identification for MBT tools according to the three standards.

<table>
<thead>
<tr>
<th>Standard</th>
<th>MBT Tool Impact According to Standard</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTCA DO-178C</td>
<td>Criteria 2</td>
</tr>
<tr>
<td>EN50128:2011</td>
<td>T2</td>
</tr>
<tr>
<td>ISO26262</td>
<td>TI1</td>
</tr>
</tbody>
</table>

**Error detection level.** MBT tools applying the replay concept described in this part are able to detect errors, so they are classified as specified in Table 23.3.

Table 23.2: Tool error detection level for MBT tools according to the three standards. Only ISO26262 specifies a well-defined level.

<table>
<thead>
<tr>
<th>Standard</th>
<th>MBT Tool Error Detection Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTCA DO-178C</td>
<td>— not applicable —</td>
</tr>
<tr>
<td>EN50128:2011</td>
<td>“tool can detect errors”</td>
</tr>
<tr>
<td>ISO26262</td>
<td>TD1</td>
</tr>
</tbody>
</table>

**Qualification levels.** For an MBT tool like RT-Tester the following qualification levels result from the previous tables. We assume that the SoS where the tool should be applied has highest criticality.

Table 23.3: Tool qualification levels for MBT tools like RT-Tester.

<table>
<thead>
<tr>
<th>Standard</th>
<th>MBT Tool Qualification Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTCA DO-178C</td>
<td>TQL4</td>
</tr>
<tr>
<td>EN50128:2011</td>
<td>no specific requirements</td>
</tr>
<tr>
<td>ISO26262</td>
<td>TCL4 for replay mechanism, otherwise no requirements</td>
</tr>
</tbody>
</table>
Qualification levels. With the assessment of MBT tools similar to RT-Tester given above, we can now summarise the applicable TQ requirements by using the strictest requirements from

- RTCA DO-178C, as far as defined for TQL4, and applicable to the whole tool,
- ISO26262, as far as defined for TCL4, but only applicable to the replay component responsible for error detection,
- EN50128:2011, as far as general documentation requirements are concerned.

The resulting TQ requirements are listed in Table 23.4. A test tool fulfilling these properties is compliant with TQ requirements of all three domains.
Table 23.4: Tool qualification requirements for MBT tools like RT-Tester, when applied to SoS testing in the automotive, avionic and railway domains.

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Enforced by Standard</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tool selection is justified by analysing its use cases.</td>
<td>[ISO09, 11.2], [CEN11, 6.7.4.2]</td>
</tr>
<tr>
<td>Effectiveness of the tool is demonstrated.</td>
<td>CEN11 6.7.4.1</td>
</tr>
<tr>
<td>Tool qualification is planned and both plan and results are documented</td>
<td>ISO09 11.4.2, [WG-11]</td>
</tr>
<tr>
<td>Tool operational requirements and functional capabilities are documented.</td>
<td>ISO09, [WG-11], [CEN11, 6.7.4.3]</td>
</tr>
<tr>
<td>Tool architecture is documented.</td>
<td>RTC11 T-2</td>
</tr>
<tr>
<td>Tool installation process is documented.</td>
<td>ISO09 11.4.2</td>
</tr>
<tr>
<td>Potential tool malfunctions have been identified and their effect on the SUT has been described</td>
<td>ISO09 11.4.2.2, CEN11 6.7.4.2</td>
</tr>
<tr>
<td>Tool operational and functional requirements are verified and checked with respect to consistency, accuracy, compatibility with the operational environment, error conditions, tool↔user interactions, verifiability, traceability and accuracy of the algorithms involved.</td>
<td>RTC11 T-3</td>
</tr>
<tr>
<td>Tool has been tested with full requirements coverage, and test results are correct / discrepancies have been explained.</td>
<td>RTC11 T-7</td>
</tr>
<tr>
<td>Tool tests are documented with test cases, procedures, expected and observed results, and with traceability data. Tests comprise normal behaviour and robustness tests.</td>
<td>RTC11 T-6</td>
</tr>
<tr>
<td>Tool configuration items are identified.</td>
<td>RTC11 T-8</td>
</tr>
<tr>
<td>Baselines and traceability are established.</td>
<td>RTC11 T-8</td>
</tr>
<tr>
<td>Problem reporting, change control, change review, and configuration status accounting are established.</td>
<td>RTC11 T-8</td>
</tr>
<tr>
<td>Archive, retrieval, and release are established.</td>
<td>RTC11 T-8</td>
</tr>
<tr>
<td>Tool life cycle environment control is established.</td>
<td>RTC11 T-8</td>
</tr>
<tr>
<td>Error detection software based on replay has been developed and verified according to RTCA DO-178B, Level A requirements.</td>
<td>ISO09</td>
</tr>
<tr>
<td>The tool interacts correctly with other tools.</td>
<td>CEN11 6.7.4.1</td>
</tr>
<tr>
<td>New tool versions have to be re-qualified.</td>
<td>CEN11 6.7.4.11</td>
</tr>
<tr>
<td>The tool life cycle and the artefacts produced during the life cycle conform to the applicable plans and standards.</td>
<td>RTC11 T-9</td>
</tr>
<tr>
<td>The person or organisation performing the TQ is documented.</td>
<td>ISO09 11.4.4.3</td>
</tr>
</tbody>
</table>
Chapter 24

Related Work

The idea of replaying a test execution in a simulator is, of course, not new. The overall approach is frequently referred to as the capture and replay paradigm, and has long been studied in different contexts such as testing of concurrent programs [CT91]. However, the classical approach of capture/replay testing is to capture user-interaction and then replay the recorded inputs within test cases, as opposed to automatic test-case generation. This paradigm differs from the one implemented in RTT-MBT, although we integrate a replay function into our work to detect deviations of a generated test case from the test model. To our best knowledge, our approach is the first to combine replay with model-based methods for error detection within a test-case generator. Our contribution is not a theoretical one, but comes from an industrial perspective. Tool qualification is compulsory for software that is applied in development processes of safety-critical systems. For ISO 26262, the tool qualification requirements in a general context were recently studied by Hillebrand et al. [HRM+11]. Most relevant to our work, the authors study verification measures for error detection, which are classified as prevention, review, and test [HRM+11, Section 4.6]. According to this classification, our approach falls into the categories review and test, as the results of the test-case generator undergo review and are automatically tested against the test model.

Recently, there has been impressive progress on verified compilers [Ler09, Myr10] that, in theory, do not require further qualification measures. França et al. [FFFL+11] report on their experiences of introducing the verified CompCert compiler into development processes for airborne software. Yet, it is important to note that even a verified compiler contains non-verified components, which may introduce bugs. Indeed, Regehr [Reg10] discovered
defects in the COMPCERT front-end responsible for type-checking, thereby showing that proofs of functional correctness of core components do not provide sufficient evidence to construct an overall correctness argument. Our approach can be seen as a practical response to this situation since we evaluate the correctness of the outputs on a per test basis.

A notable difference between ISO 26262 and RTCA DO-178C is that the latter standard distinguishes between tools that mutate the software (such as code generators) and those that only analyze it (such as stack analyzers). The qualification measures imposed onto analyzers or test-case generators are less strict in RTCA DO-178C. Further details on this issue in the context of RTCA DO-178C (that are likewise applicable to DO-178B), and also additional information about the qualification process for formal verification tools, are given by Souyris et al. [SWDD09]. This recent paper can be seen as a wrap-up of a paper by the same authors that studied the same problem more than a decade ago [RSB+99], and also contains details about how different formal methods tools are used within Airbus. Qualification for RTCA DO-178B was mentioned earlier by Blackburn and Busser [BB96]. There, the authors describe the tool T-Vec, which is used in the qualification process to test itself. However, rather than using automatic replay, they manually define expected outputs and compare them to derived test procedures [BB96, Section 5]. By way of contrast, our approach delegates the manual specification of expected outputs to the design of the test model — a step that is necessary in model-based testing anyway — and the execution of test procedures.
Chapter 25

Concluding Discussion

This report advocates the use of replay for tool error detection in model-based test generators as a key mechanism for qualifying such a tool according to ISO 26262, since it provides trustworthy, yet simple and cost effective, error detection. By providing full tool error detection capabilities, the approach thus smoothly integrates with the requirements of ISO 26262 for the highest tool confidence level. It is noteworthy, however, that the mechanism does not ensure the absence of errors in the entire tool, but only in the functionality that is indeed used.

We have to point out, however, that the qualification achieved for ISO 26262 cannot be directly used to gain qualification credit according to RTCA DO-178C: the latter standard requires comprehensive requirements-based testing of all tool capabilities, while being less strict with respect to the verification of error detection mechanisms, which are considered simply as tool capabilities to be verified by means of robustness tests. The verification techniques to achieve this are the same as the ones used for qualification according to ISO 26262.
Part V

Conclusion and Outlook
Chapter 26

Conclusion

In this technical report the state of the art and the specific challenges and current and future research fields for testing SoS have been described. Model-based testing appears to be the most promising methodology for systematic testing in an SoS context. The SysML modelling language is suitable for specifying complex distributed systems; therefore it is an obvious candidate for designing test models for SoS system testing. We have extensively described the COMPASS MBT approach with the test automation tool RT-Tester for testing against SysML test models. This approach comprises automated test case identification, automated test data and test procedure generation, as well as automated requirements tracing, and all automation techniques are based on exploiting either abstract syntax or operational semantics of the SysML model.

Since the test model is the pre-requisite for all MBT activities, its correctness and completeness is of specific importance to SoS V&V. Therefore bounded model checking support is provided as an integrated function of RT-Tester. Further techniques for model V&V, such as simulation and mutation generation, are also available in RT-Tester; they will be described in the final version of this report at the end of the project.

As an alternative to SysML, the textual variant of the COMPASS modelling language CML can be used. For textual CML, the basic construction techniques for generating the transition relation have been specified. In the next project phases the CML support will be extended; this will be described in a new release of the present document.

The automated MBT technology as described in the current baseline of this technical report focused on SysML test model support for complex distributed
real-time systems, and on integration of RT-Tester into the COMPASS tool platform. The results presented here are well-suited to

- perform comprehensive testing on constituent system level, and
- test certain aspects of complete SoS that only exhibit “mild” variants of non-determinism.

In this document version we do not yet cover the more advanced aspects of SoS testing which have been described in the first part of this document, such as

- SoS-specific test strategies and their formal justification,
- contract representation for test purposes,
- re-use of test results obtained for constituent systems,
- SoS test model elaboration based on constituent system models with different meta-meta models,
- incomplete test models and resulting “strong” variants of non-determinism,
- dynamic object creation and destruction,
- on-the-fly contract negotiation and run-time testing against these contracts,
- incremental test model creation.

Approaches for coping with these challenges have been elaborated in the first year of the COMPASS project, and will be documented in the next release of this document and in

- Technical Report Number D34.2: Specialised Test Strategies.

Some of the related activities to be performed next are sketched in the next chapter.
Chapter 27

Outlook on COMPASS SoS Test Automation

The SoS-specific test-related challenges listed in the previous chapter require specialised test strategies and their formal justification. The specialisations investigated in COMPASS will focus on

- state space size and complexity of data structures,
- dynamic adaption of test objectives to changes in SoS configuration and contract conditions.

The size of SoS state spaces implies that exhaustive investigation of the complete concrete state space will certainly be infeasible. We tackle this problem by two orthogonal strategies.

- On constituent system level, different behaviours associated with the same local mission threads will be comprised in equivalence classes. This reduces the complexity problem for SoS system testing to covering combinations of classes of constituent system behaviours instead of sequences of concrete state vector combinations.

- On SoS system level, “relevant” class combinations are identified by means of different variants of impact analysis, such as data flow analyses or investigation of contractual dependencies. Behaviours of constituent systems which do not affect the relevant class combinations under consideration will be selected according to the principle of orthogonal arrays.
For coping with complex data structures the SONOLAR SMT solver is already equipped with support for floating point arithmetics and a theory of arrays. These capabilities will be exploited in the next phase of the project to support complex data structures like lists of lists and structured elements, as already defined in the CML. Since testing always considers finite traces for stimulating the SUT and observing SUT reactions, all dynamic data structures may be considered to be bounded: these structures can only change by a bounded amount of elements on a bounded sequence of states. As a consequence, problems well-known from infinite-state model checking will not occur in the context of MBT.

The dynamic adaptation of test objectives is particularly relevant for run-time acceptance testing of changing SoS configurations. In contrast to development models for SoS, however, we only have to consider bounded changes of SoS configurations and contractual boundary conditions, because every test suite can only consider a bounded number of configurations anyway. It remains to investigate how to determine configurations possessing sufficient error detection strength. Results from the field of mutation testing will help to determine this strength in a systematic and measurable way.

Incomplete test models will be considered in the following way.

- The SoS system behaviour is structured into several top-level operational modes. It is expected that switching between these modes can be performed in a deterministic way for normal behaviour tests: it is unlikely that SoS performing operational mode changes only on a random basis are acceptable and “testworthy”.

- Entry into failure modes is non-deterministic, but can be initiated in a deterministic way for test purposes by means of pre-planned failure injections.

- The behaviour in each operational mode is not completely deterministic, but can be captured by sets of constraints governing the acceptable computations in each mode. Test oracles will therefore no longer check for explicit output traces of the SUT but for compliance of the traces observed with the constraints applicable in each mode.

- For test stimulation purposes the SMT solver computes sequences of feasible mode switches and the test data provoking these switches.

- Incremental test model elaboration can be performed by adding constraints identified during test observations to the modes where they are applicable. To this end, techniques from machine learning seem to be
promising.

Justification of test strategies will be performed by proving that they will “converge” to exhaustive tests proving some compliance relation between SUT and reference model. Though execution of exhaustive test suites will generally be infeasible in practice, convergence to exhaustive test suites ensures that new test cases added to the suite will really increase the assurance level by a positive amount: intuitively designed test strategies often do not possess this property, because additional test cases may just re-test SUT aspects already covered by existing ones.
Bibliography


