Quality Assurance of Test Specifications for Reactive Systems

Dissertation

zur Erlangung des Doktorgrades
der Mathematisch-Naturwissenschaftlichen Fakultäten
der Georg-August-Universität zu Göttingen

vorgelegt von

Benjamin Immanuel Eberhardt Elmar Zeiß
aus Frankfurt am Main

Göttingen
im Mai 2010

Referent:	Prof. Dr. phil. nat. Jens Grabowski, Georg-August Universität Göttingen.
Korreferentin:	Prof. Dr.-Ing. Ina Schieferdecker, Technische Universität Berlin.

Tag der mündlichen Prüfung: 2. Juni 2010

Abstract

Extensive testing of modern communicating systems often involves large and complex test suites that need to be maintained throughout the life cycle of the tested systems. For this purpose, quality assurance of test suites is an inevitable task that eventually may have an impact on the quality of the system under test as well.

In this thesis, we present a holistic method towards the analytical quality engineering of test specifications. We cover in detail what constitutes the quality of test specifications by adapting a quality model for software to test specifications and present how to apply target-oriented static testing to test specifications. We also introduce a dynamic testing method for test specifications, including a reverse engineering approach for test behavior models, and present a method for the consistency analysis of system responses in test suites. Based on the quality assessments made, the test suites can be improved regarding specific quality characteristics of this quality model. Finally, we validate and demonstrate the applicability of our approaches in a case study by means of a prototype implementation.

Zusammenfassung

Umfassendes Testen von modernen kommunizierenden Systemen beinhaltet oft große und komplexe Testsuiten, die über den Lebenszyklus des getesteten Systems hinweg gewartet werden müssen. Daher ist Qualitätssicherung von Testsuiten eine unvermeidliche Aufgabe, die letztendlich auch einen Einfluss auf die Qualität des getesteten Systems haben kann.

In dieser Arbeit wird eine ganzheitliche Methode zur analytischen Qualitätssicherung von Testspezifikationen vorgestellt. Sie beschreibt die Qualität von Testspezifikationen, indem ein allgemeines Qualitätsmodell für Softwareprodukte adaptiert wird. Es wird gezeigt wie statisches Testen zielgerichtet auf Testspezifikationen angewendet werden kann und wie dynamisches Testen von abstrakten Testspezifikationen möglich ist. Für den dynamischen Testansatz wird eine Reverse-Engineering Methode zur Gewinnung von Test-Verhaltensmodellen beschrieben sowie eine Konsistenzanalyse für Systemantworten in Testsuiten diskutiert. Basierend auf Qualitätsbewertungen können Testsuiten bezüglich spezifischer Qualitätscharakteristiken des Qualitätsmodelles verbessert werden. Zuletzt werden die präsentierten Ansätze in einer Fallstudie validiert und Ihre praktische Anwendbarkeit mit Hilfe einer prototypischen Implementierung gezeigt.

Acknowledgements

A lot of people had influence on me and my work during my PhD studies in various different ways. All of them have played in some way a role in the past few years of my life. Therefore, it is a pleasure for me to at pay my respects with this small section. People are listed in no particular order.

First, I like to thank my mentors Prof. Dr. Jens Grabowski, who is also my primary supervisor, Dr. Andreas Ulrich, my external advisor, and Dr. Helmut Neukirchen, my former colleague, supervisor of my master's thesis, and as well a member of my thesis committee, for their valuable support during my research. Their knowledge, suggestions, and ideas contributed a lot to the results of this thesis and I never ceased to learn from their comments up to the present day. In addition, without the support of the Siemens AG, this work would not have been possible.

I would like to thank Prof. Dr. Ina Schieferdecker for accepting the co-review of my thesis. My colleagues in Göttingen always provided an enjoyable and scientifically inspiring work environment: Thomas Rings, Philip Makedonski, Steffen Herbold, Edith Werner, Wafi Dahman, Akthar Ali Jalbani, Gunnar Krull, and Annette Kadziora. I am especially grateful for the efforts that Thomas Rings and Philip Makedonski put into the proof reading of this thesis. I would like to thank the remaining members of my defense committee: Prof. Dr. Stephan Waack, Prof. Dr. Carsten Damm, Prof. Dr. Dieter Hogrefe, Prof. Dr. Xiaoming Fu.

I would like to express my gratitude to the following people who I think should be mentioned here: Klaus Beetz, Horst Sauer, Peter Zimmer, Guy Collins Ndem, Dr. Joachim Fröhlich, Thanik Cheowtirakul, and Dennis Neumann. I am sorry if I forgot anyone.

My family has always supported me unconditionally during my studies: Werner, Gisa, Daniel, Eva, Sophia, Alessia, and Liselotte. In particular, I would like to thank my uncle Michael, my aunt Andrea, and my cousins for the time they spent with me during my visits in Munich.

Contents

1. Introduction **1**
1.1. Software Quality Assurance . . . 1
1.2. Distinctive Properties of Test Specifications . . . 4
1.3. Quality Assurance for Test Specifications . . . 6
1.4. Thesis Scope, Approach, and Contribution . . . 8
1.5. Impact . . . 10
1.6. Thesis Structure . . . 11

2. Prerequisites **13**
2.1. Testing and Test Control Notation 3 (TTCN-3) . . . 13
2.1.1. Distributed Testing . . . 13
2.1.2. Templates . . . 14
2.1.3. Behavior, Alternative Behavior, and Defaults . . . 15
2.1.4. Verdict Mechanism . . . 16
2.1.5. Abstraction . . . 16
2.1.6. Example . . . 17
2.2. Formal Notations for Transition Systems . . . 18
2.2.1. Formal Models . . . 18
2.2.2. Parallel Composition . . . 21
2.2.3. Queues . . . 22
2.2.4. Implementation and Equivalence Relations . . . 25
2.3. Model Checking . . . 26
2.3.1. Temporal Logic . . . 27
2.3.2. Properties . . . 28
2.3.3. Property Patterns . . . 28
2.3.4. Analysis Classification . . . 29

3. A Quality Model for Test Specifications **30**
3.1. Software Quality Models . . . 31
3.1.1. The ISO/IEC 9126 Standard . . . 32
3.1.2. The FUPRS, FURPS+, and Dromey Quality Models . . . 33
3.1.3. Related Software Quality Standards . . . 33

3.2. Adapting the ISO 9126 Quality Model for Test Specifications 33
3.2.1. New Characteristics 35
3.2.2. Changed Characteristics 35
3.2.3. Removed Characteristics 35
3.3. Quality Characteristics of the Quality Model for Test Specifications 36
3.3.1. Test Effectivity 37
3.3.2. Reliability 38
3.3.3. Usability 39
3.3.4. Efficiency 40
3.3.5. Maintainability 40
3.3.6. Portability 41
3.3.7. Reusability 41
3.4. Towards an Instantiation of the Quality Model for Test Specifications 42
3.4.1. Software Metrics 42
3.4.2. Software Metrics Selection 44
3.4.3. Bad Smells 46
3.4.4. Refactoring 47
3.4.5. Static Analysis 47
3.4.6. Dynamic Analysis 48
3.5. An Instantiation for TTCN-3 Projects 49
3.5.1. Characteristic: Test Effectivity / Test Coverage 50
3.5.2. Characteristic: Test Effectivity / Test Correctness 51
3.5.3. Characteristic: Test Effectivity / Fault-Revealing Capability 52
3.5.4. Characteristic: Test Reliability / Maturity 53
3.5.5. Characteristic: Maintainability / Analyzability 54
3.5.6. Characteristic: Maintainability / Changeability 54
3.5.7. Characteristic: Maintainability / Stability 56
3.5.8. Characteristic: Reusability / Coupling 56
3.5.9. Characteristic: Reusability / Flexibility 57
3.5.10. Characteristic: Reusability / Comprehensibility 58
3.5.11. Characteristic: Compliance 59
3.6. Related Work 59

4. Model-Based Analysis of Test Specifications 62
4.1. Reverse Engineering 63
4.1.1. Documentation and Design Recovery 64
4.1.2. Reengineering 64
4.1.3. Recovering and Reengineering Artifacts of Development Phases . . 64
4.2. A Formal Model for Describing Test Case Specifications 65
4.2.1. Test Case Model 66
4.2.2. Test Case Model Example 67

4.2.3. Test Case Model Adequacy and Abstraction 68
4.3. Model Reconstruction 70
4.3.1. Logging 72
4.3.2. Behavior Model Reconstruction 75
4.3.3. Test Case Simulation 78
4.4. Test Case Analysis 80
4.4.1. Test-Specific Anomalies 81
4.4.2. TTCN-3 Specific Anomalies 85
4.4.3. Generic Anomalies 89
4.4.4. Limitations 96
4.5. Response Consistency 96
4.5.1. Introductory Example 97
4.5.2. Response Consistency Definition 99
4.5.3. Response Consistency Detection Algorithm 101
4.5.4. Scenario: Sequential Models with Different Response Orders . . . 102
4.5.5. Scenario: Concurrent Models 104
4.5.6. Limitations 107
4.6. Related Work 108

5. Applied Automated Test Quality Assessment 111
5.1. Test Specification Analyzer Implementation 111
5.1.1. Static Analysis and Refactoring 112
5.1.2. Model-Based Analysis 113
5.2. Case Study 123
5.2.1. The ETSI SIP, IPv6, and HiperMAN Test Suites 123
5.2.2. Static Assessment and Improvement 124
5.2.3. Model-Based Assessment of Dynamically Detectable Properties . . 126

6. Conclusion 134
6.1. Summary 134
6.2. Outlook 135

Bibliography 139

A. Appendix 151
A.1. Model Reconstruction Algorithm 151
A.2. Model Reconstruction Example 154
A.3. Simulation Coloring 161
A.4. The LTSML Metamodel 162

List of Figures

1.1. Software Quality Assurance Overview . . . 2
1.2. Analytical Quality Assurance . . . 3
1.3. ISTQB Fundamental Test Process . . . 7
1.4. The Analytical Quality Assurance Cycle . . . 9

2.1. TTCN-3 Test System Architecture . . . 14
2.2. TTCN-3 System Structure . . . 16
2.3. Test System Input Queue . . . 23
2.4. Queue of Length 2 Accepting Messages r and s . . . 24

3.1. Quality Model Adaptation Procedure . . . 31
3.2. ISO 9126 Quality Model . . . 32
3.3. ISO 9124 Adaptation for Test Specifications . . . 34
3.4. Instantiated Parts of the Quality Model for Test Specifications . . . 50

4.1. Dynamic Analysis Methodology . . . 62
4.2. Forward and Reverse Engineering in the V-Modell 97 Phases . . . 65
4.3. Example: TTCN-3 Test Case to EMIOTS Component Mapping . . . 67
4.4. Example: Mapping Log Events to a Model . . . 74
4.5. Test Behavior Simulation . . . 79
4.6. Example: Missing Test Verdict . . . 81
4.7. Example: Fail/Inconc Verdict Decision Before Communication . . . 83
4.8. Example: Verdict Consistency . . . 84
4.9. Example: Idle PTC . . . 86
4.10. Example: Altstep Asymmetry . . . 87
4.11. Example: Send/Receive on Unconnected/Unmapped Ports . . . 88
4.12. Example: Data-Flow Anomalies . . . 90
4.13. Example: Deadlock . . . 92
4.14. Example: Livelock . . . 93
4.15. Example: Illegal Double Calls . . . 95
4.16. Example: Response Inconsistency . . . 98
4.17. Local Test Cases with Different Response Orders . . . 103
4.18. Model T_{10} . . . 105
4.19. Model T_{11}: Composite Model T_{10} Without Synchronization . . . 106

5.1. TRex Tool Chain 113
5.2. Tool Workflow 114
5.3. Example: Verdict before Communication (Dot Visualization) 120
5.4. LOC Before and After Applying Refactorings 125
5.5. Number of Templates Before and After Applying Refactorings 126

A.1. Reconstructed Model 160
A.2. LTSML Metamodel 163

List of Tables

4.1. Log Consistency Rules 75
4.2. Example: a Test Run Log λ 76

5.1. Size of ETSI Test Suites 124
5.2. The Analyzed Test Cases 128

A.1. Example Log 1 155
A.2. Example Log 2 157

Listings

2.1. Example: TTCN-3 Template . . . 15
2.2. Example: TTCN-3 Code . . . 17

5.1. Example: Verdict Before Communication (TTCN-3) . . . 116
5.2. Example: Verdict before Communication (LTSML) - Part 1/6 . . . 117
5.3. Example: Verdict before Communication (LTSML) - Part 2/6 . . . 117
5.4. Example: Verdict before Communication (LTSML) - Part 3/6 . . . 117
5.5. Example: Verdict before Communication (LTSML) - Part 4/6 . . . 118
5.6. Example: Verdict before Communication (LTSML) - Part 5/6 . . . 118
5.7. Example: Verdict before Communication (LTSML) - Part 6/6 . . . 119
5.8. Example: Verdict before Communication (Promela) - Part 1/3 . . . 119
5.9. Example: Verdict before Communication (Promela) - Part 2/3 . . . 121
5.10. Example: Verdict before Communication (Promela) - Part 3/3 . . . 122

Acronyms

ANTLR *ANother Tool for Language Recognition*

ATS *Abstract Test Suite*

BRAN *Broadband Radio Access Network*

CCS *Calculus of Communicating Systems*

CMMI *Capability Maturity Model Integration*

CSP *Communicating Sequential Processes*

CTL *Computation Tree Logic*

CTMF *Conformance Testing Methodology and Framework*

CTP *Critical Testing Processes*

DLC *Data Link Control*

DOM *Document Object Model*

DSL *Domain Specific Language*

EFSM *Extended Finite State Machine*

EMF *Eclipse Modeling Framework*

EMIOTS *Extended Multi-Input/Output Transition System*

ETSI *European Telecommunications Standards Institute*

FCM *Factor-Criteria-Metrics*

FIFO *First In, First Out*

GQM *Goal Question Metric*

HiperMAN *High Performance Metropolitan Area Network*

IDE *Integrated Development Environment*

IETF *Internet Engineering Task Force*

IPv6 *Internet Protocol Version 6*

ISTQB *International Software Testing Qualifications Board*

JAXB *Java Architecture for XML Binding*

JDT *Java Development Tools*

JMS *Java Message Service*

LOC *Lines of Code*

LTE *Long Term Evolution*

LTL *Linear Temporal Logic*

LTS *Labeled Transition System*

LTSML *Labeled Transition System Markup Language*

MOM *Message Oriented Middleware*

MTC *Main Test Component*

OSI *Open Systems Interconnection*

PTC *Parallel Test Component*

RFC *Request for Comments*

RUP *Rational Unified Process*

SAX *Simple API for XML*

SIP *Session Initiation Protocol*

SOS *Structural Operational Semantics*

STEP *Systematic Test and Evaluation Process*

SUT *System Under Test*

TCI *Test Control Interface*

TCTL *Timed Computation Tree Logic*

TMMi *Test Maturity Model Integrated*

TPI *Test Process Improvement*

TRex *TTCN-3 Refactoring and Metrics Tool*

TRI *Test Runtime Interface*

TSI *Test System Interface*

TTCN *Tree and Tabular Combined Notation*

TTCN-3 *Testing and Test Control Notation*

U2TP *UML 2.0 Testing Profile*

UML *Unified Modeling Language*

WiMAX *Worldwide Interoperability for Microwave Access*

XML *Extensible Markup Language*

XSLT *Extensible Stylesheet Language Transformations*

1. Introduction

Industrial test suites for modern communicating systems are often huge in size and complex in behavior. The tested devices and systems are becoming increasingly sophisticated and at the same time, they have to be more reliable than ever before. Nowadays, extensive testing involves not only the specification, the selection, and the execution of test cases, but also includes the maintenance of test suites throughout the life cycle of the tested system. For this purpose, quality assurance of test suites is an inevitable task that eventually may have an impact on the quality of the *System Under Test* (SUT). There is always a reciprocal effect between the quality of the test suite and the quality of the SUT. In addition, just like general-purpose software, test suites suffer from software aging [116]. It is thus sensible to find quality issues in tests as early as possible. For that purpose, we need to apply quality assurance and quality assurance techniques to test suites as well.

1.1. Software Quality Assurance

Quality assurance for general-purpose software is nothing new—in fact, it is a main subject in the field of software engineering. The IEEE 610 standard [108] defines the term *quality* as the "degree to which a system, component, or process meets specified requirements" and the "degree to which a system, component, or process meets customer or user needs or expectations". The overall fitness of such a software product is usually determined by its internal characteristics (i.e., characteristics derived from an internal view of the software product, such as code or specification documents that can be improved during implementation, reviewing, and testing), its characteristics when it is executed (external quality), and the characteristics that it exposes when it is used. These characteristics are weighted according to the needs, requirements, and expectations of this project. The needs towards the quality characteristics of two different software products are rarely the same. In the end of 2005, the Tokyo Stock Exchange had to deal with various software issues [135]. Among them was an issue where the cancel command for trades failed. An employee of Mizuho Securities mistakenly typed to sell 610,000 shares at 1 Yen instead of one share at 610,000 Yen. Even though Mizuho noticed the mistake in time, they were unable to cancel the order due to the bug in the software system denying the cancel command. Mizuho's loss amounted to approximately 225 million US dollars. Being only the most prominent example of loss caused by software problems at the Tokyo Stock Exchange in late 2005, it is likely that the total sum of losses due to software glitches is a lot higher. On the other

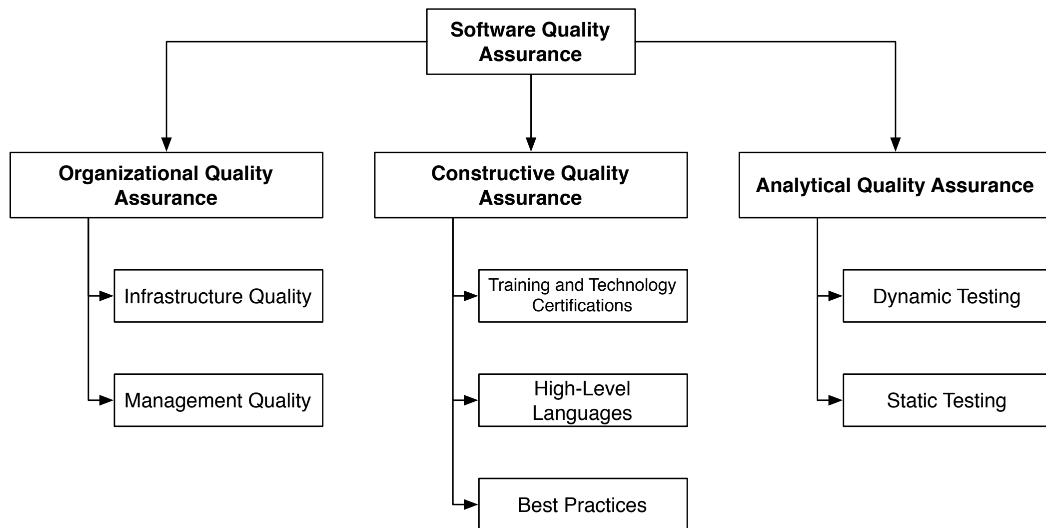


Figure 1.1.: Software Quality Assurance Overview

hand, a crashing word processing application may (in most cases) only cause productivity loss for an individual person in the worst case. Quality depends on many non-technical factors such as time, cost, size of the development team, developer experience. It can be decisive for the success of a software product in the non-critical case, whereas for critical software, software quality is a factor that may prevent or trigger catastrophes.

To make sure that software quality characteristics are respected during development and within maintenance of a software product, quality assurance measures need to be established. The IEEE 610 standard defines quality assurance as a "planned and systematic pattern of all actions necessary to provide adequate confidence that an item or product conforms to established technical requirements". A second definition in the standard states that quality assurance is a "set of activities designed to evaluate the process by which products are developed or manufactured". This twofold definition stresses that software quality assurance concerns not only the evaluation of the software product, but also the software development process and its procedures. Software quality assurance can be viewed from a multitude of different angles. Different factors within the software development process influence quality characteristics of a software product. These factors are not only found in the analysis, design, specification, and implementation of the software product, but also in the organizational parts of a project of technical and non-technical nature that surround the actual software development.

Figure 1.1 illustrates an overview how quality assurance can be subdivided [78]. Organizational quality measures concern the organization of the infrastructure and the management. Means for supporting the infrastructure are configuration management or defect management. Management quality is influenced by process models and process improve-

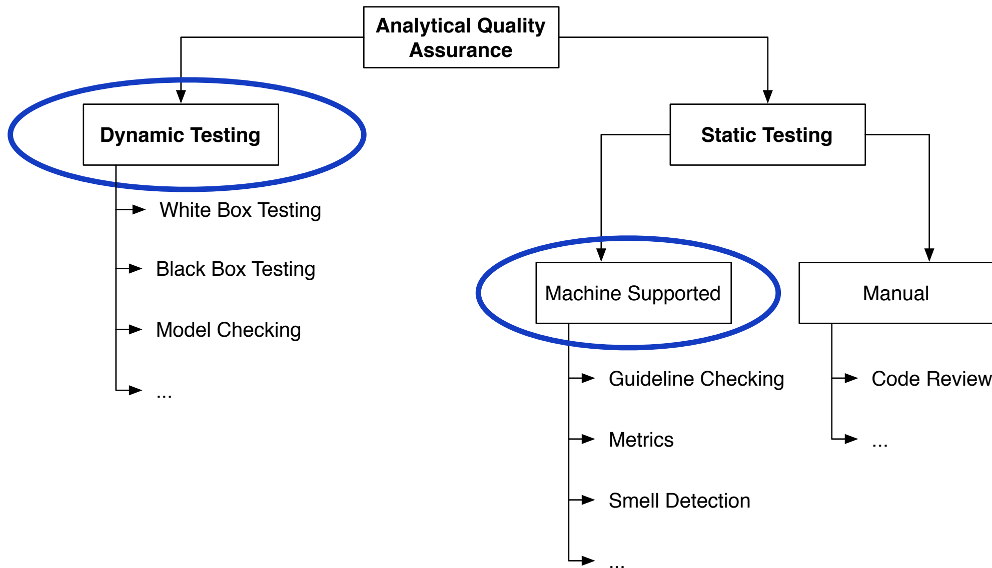


Figure 1.2.: Analytical Quality Assurance

ment models. Process models comprise models such as the V-Modell 97 [60], iterative models such as *Rational Unified Process* (RUP) [94], or incremental and agile models such as Extreme Programming [11]. Process improvement models comprise models such as *Capability Maturity Model Integration* (CMMI) [26, 27, 28] or the ISO/IEC 15504 standard [89]. They define criteria based on which a product can claim a certain level of maturity.

Constructive quality assurance comprises activities that prevent future quality problems by avoiding issues. These measures are proactive and often start before the actual project begins. Constructive measures are technical trainings, certifications, choice of tooling, formalisms that improve communication (for example, by using *Unified Modeling Language* (UML) [111, 112] for documentation), or usage of best practices such as design patterns [65] or refactoring [64].

Analytical measures, the primary focus of this thesis, are reactive to existing problems and try to locate and eliminate quality problems when specifications or code has been produced already. Figure 1.2 illustrates how analytical quality assurance can be structured. The vocabulary and the differentiation between static testing and dynamic testing are aligned to definitions given by the *International Software Testing Qualifications Board* (ISTQB) [134]. Static testing is applied when the software is not actually used or executed. Rather, it essentially means to look at a software product and possibly its underlying code. Static testing can be further subdivided into machine supported testing and manual testing. Machine supported static testing is applied when the specification (or code at a lower level) of the soft-

ware product is automatically analyzed in some manner. This may include simple semantic checks of a compiler, but it may also include the checking of guidelines, the calculation of *software metrics* [61], or the detection of *bad smells* [64]. Manual static testing, on the other hand, is performed without any kind of machine support and includes, for example, manual technical reviews or code inspections. Once possible issues have been identified and located, methods for improvement can be applied.

1.2. Distinctive Properties of Test Specifications

To discuss quality assurance for test specifications, it is necessary to identify the differences between a test specification and a general-purpose software product. This is especially of particular importance since we use and present methods that are already known for general-purpose software systems, but adapted and instantiated for the context of test specifications.

A test specification is an abstract document that specifies a *test suite* with the purpose to test another software system, the SUT, in order to find bugs or misbehaviors. A test suite itself is composed by a number of *test cases*. It describes the behavior of test cases, the interfaces of the SUT, the execution orders of the tests, data types, and test data descriptions. The degree to which a test specification is formalized can differ significantly. An informal test specification could be a document written in a text processor in a natural language with instructions how to manually perform the tests in the test specification. A formal test specification could be a specification written in a language like *Testing and Test Control Notation* (TTCN-3), they could be described in the form of state machines, or in the form of a UML model. There are clear differences between those more formal test specifications. The language TTCN-3, for example, has strict syntax and semantics. However, the overall complexity of the language is relatively high in both its grammatical and semantic nature. A state machine on the other hand can represent test behavior using simple syntax and semantics. However, it is not as expressive and as high-level as a language like TTCN-3. Tests specified using UML models, e.g., using the *UML 2.0 Testing Profile* (U2TP) [110], may appear easy to understand due to its graphical representation, but they lack the kind of strict semantics that a language such as TTCN-3 offers. UML does not exactly state how and when to use its diagrams (or model elements) and how they can or must be connected to each other. Therefore, models are likely to look different unless they are forced to follow specific custom guidelines.

Having discussed the term *test specification* and what its requirements are, we continue the discussion about the differences between a software system and a test specification. The goal of a test case for communicating systems is to produce a concrete answer to a specific question. The question is whether a certain behavior executes as expected, i.e., the test simulates a specific communication scenario between the SUT and its environment. The test replaces the environment and simulates its behavior for this scenario. Depending on the SUT reactions, the test case concludes with an answer—a verdict about the tested

scenario. Using a set of such scenarios with associated conclusions, a certain degree of security towards the quality of the tested aspect of the SUT is assumed. While tests do not assure any kind of correctness towards the tested aspect, they do help to build confidence in the system that is tested.

In order to reproduce the scenario that has been tested, the test should ideally be deterministic, i.e., it should be able to repeat its behavior in exactly the same way as before. This is important since missing repeatability can make a problem hard to debug and fix when the problem cannot be reproduced easily and reliably. Therefore, any kind of randomness introduced into the test behavior, either explicitly, by using random decisions, or implicitly, for example, due to the specification of parallel behavior that may be scheduled differently in each execution, can cause trouble in the analysis of the underlying problem.

Test specifications are usually designed to have as little user interaction as possible. Often, there is no user interaction at all. The reason is that the unsupervised execution of instantiated test specifications is on the one hand more effective than any kind of supervised execution and on the other hand the confidence and security towards the tested system increases the more often the tests are executed—due to the expectation that the automated test cases catch a high number of mistakes introduced due to changes. Depending on the concrete test, however, full automation may not be always possible or feasible.

Finally, one very distinctive property of test specifications is that they are usually not tested dynamically. Even though test specifications are software artifacts just like the software product itself, it is not considered to be feasible to write test cases for test specifications as well. After all, the tests that check the test specifications would strictly have to be tested again. In addition, system tests in particular are often defined against design documents and interface specifications when the actual SUT is still in development. Thus, there is often no possibility to execute the test specifications against its target. As a result, the quality assurance measures applied to general-purpose software, especially dynamic testing techniques, need to be adapted for test specifications to fit their requirements.

To summarize, the main distinctive properties of test specifications as opposed to general-purpose software systems are the following:

- The purpose of a test specification is to describe behavior that evaluates certain aspects of the SUT. It concludes with a test verdict.
- Tests should ideally be repeatable. To achieve that, their behavior should ideally be deterministic.
- Depending on the test type, test behavior is often designed to be executed unsupervised and without any user interaction. Effortless execution of test behavior is a requirement for a continuous safety net that may catch errors due to changes.
- Test specifications cannot be tested in the same way. Quality assurance measures therefore need to be adapted.

1.3. Quality Assurance for Test Specifications

In general, the work on quality assurance for test cases or test suites is rare despite the fact that they present software products as well. As an example, a 3GPP *Long Term Evolution* (LTE) test suite for mobile terminals currently in development at the *European Telecommunications Standards Institute* (ETSI) already encompasses over 160,000 lines of code. Test suites can be not only huge in size, but they are also reasonably complex. Therefore, means for quality assurance of test specifications is an inevitable necessity.

In the area of organizational quality assurance, the infrastructure quality measures can be applied to test specifications in a similar way as they are applied to general-purpose software. For the management quality assurance, there are several test process improvement models available, such as *Test Maturity Model Integrated* (TMMi) [138], *Test Process Improvement* (TPI) [93], *Critical Testing Processes* (CTP) [19], or *Systematic Test and Evaluation Process* (STEP) [38]. For test processes, there is, for example, the fundamental test process from the ISTQB [21].

Among the constructive measures are, for example, certifications and trainings for the certified tester program from the ISTQB [20], that are currently available at the foundation level and advanced level. High-level languages can be a constructive measure by reducing the potential for errors due to a high degree of abstraction and domain-specific features—a high degree of abstraction makes test specifications easier to understand and thus less error-prone. The introduction of domain-specific features can make the specification more natural (in the case of test specifications, for example, verdict handling would be such a domain-specific feature). TTCN-3 is such a domain-specific high-level language for test specifications. Best practices for the specification of tests exist in the form of pattern catalogs for TTCN-3 [150] or smell and refactoring descriptions for TTCN-3 and xUnit test specifications [103, 154].

Analytical measures regarding test specifications primarily take place in the form of manual reviews. Often, manual reviews are the only available possibility to evaluate informal design documents or test descriptions written in natural language. Work on machine-supported analytical measures for the assessment and improvement of test specifications on the other hand is relatively rare (see Sections 3.6 and 4.6). There is currently no work that instantiates quality assurance for test specifications with a systematic approach regarding the classification, identification, and removal of possible quality problems in test specifications. In particular, a feasible solution for machine-supported dynamic testing of test specifications has not been presented so far.

Enforcing systematic quality assurance of test specifications, no matter whether organizational, constructive, or analytical, affects the test process in all its phases. Figure 1.3 illustrates the ISTQB fundamental test process [20]. It contains six phases in which the test process is partitioned and one phase spanning over the other six, the control phase. In the planning phase, resources are allocated and the test strategy is chosen, for example, to decide which system parts are critical to test and which are not. In the test analysis, test

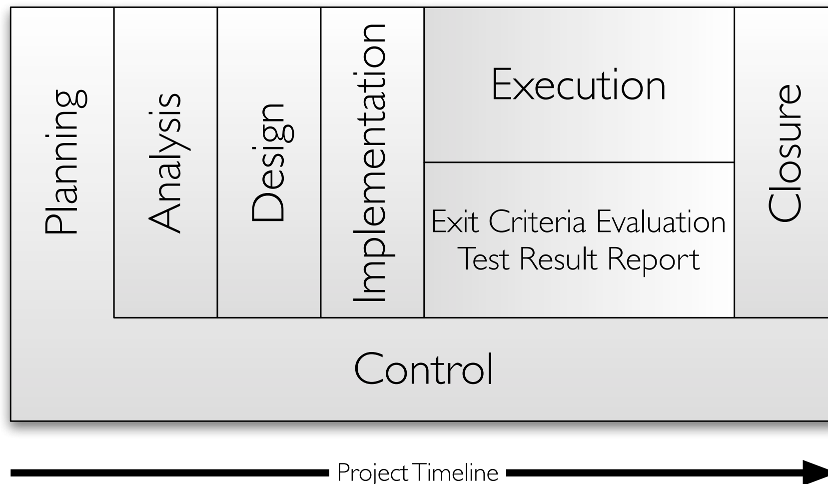


Figure 1.3.: ISTQB Fundamental Test Process

design, and test implementation, logical test cases (without concrete test input data) and afterwards concrete test cases are defined according to the selected test strategy. The test implementation includes the establishment of the test infrastructure and, in general, all necessary steps to make the tests executable. In the execution phase, the test cases are executed against the SUT and the involved steps are journalized. If a test case fails, the journal, the test environment, and the input data help to make the possible problem reproducible. These artifacts that result from the execution are evaluated and then passed on to the responsible persons in the exit criteria evaluation and test result report phase. In the closure phase, the results of the previous phases are examined and prepared for the use in other projects so that the lessons that have been learned are not lost and can be made useful for other projects or in new iterations of the same project that may take place. Finally, the control phase makes sure that current test activities are executed according to the test plan and according to the resources allocated. If deviations are detected, test control commences the necessary steps achieve the test plan goals.

Quality assurance of the test specifications should take place in all phases of the test process. Constructive and organizational quality assurance primarily takes place in the planning phase. For example, the necessary infrastructure is allocated, test developers are chosen or sent to technical trainings, and technical decisions are made that ensure high quality artifacts later on. Analytical quality assurance, on the other hand, takes place in all phases of the test process. All artifacts, even the artifacts describing the resource allocation or the test strategy should be reviewed. Therefore, in a test process with enforced quality assurance, static testing takes place in all phases to ensure that the documents are correct. Automated static and dynamic testing can be primarily performed in the analysis, design, implementation,

and execution phases. The execution phase and test evaluation phases can be considered to be supportive in testing the test specification as the examination of the test results may also uncover errors or anomalies in the test specification. In that sense, the test specification tests the SUT and vice-versa. The artifacts produced in the analysis, design, and implementation phases can be automatically checked statically or even tested dynamically. The primary purpose of this thesis is to demonstrate how exactly to perform testing on test specifications that result from the analysis, design, and implementation phase.

While we can differentiate between internal and external quality characteristics for test specifications as well, external characteristics only become apparent when the tests are executed. However, test specifications by themselves are abstract on the one hand and on the other hand, we want to concentrate on quality assurance during the development, i.e., when the actual system might not be available for testing. Thus, we concentrate on internal quality characteristics of test specifications in the following.

1.4. Thesis Scope, Approach, and Contribution

We concentrate on those items in the analytical quality assurance that have not been covered holistically for test specifications, i.e., we cover in detail what quality for test specifications exactly constitutes and based on that, we survey how to apply machine-supported static testing to test specifications with the help of software metrics and bad smell patterns. We introduce a dynamic testing method for test specifications that is feasible to apply and present a method for the consistency analysis of responses in test suites. These items are covered with regard to high-level message-based black-box test specifications with queues. The intent of this thesis is to show how *systematic machine-supported analytical quality assurance using both static testing and dynamic testing, respecting test-specific characteristics, is possible and feasible*.

The holistic approach of this thesis is the application of the presented quality assurance techniques as part of the test development process in a quality assurance cycle (illustrated in Figure 1.4). The cycle consists of four phases. In the first phase, a quality characteristic that should be assessed is chosen. The quality characteristic is derived from an adapted quality model for test specifications. Based on the chosen quality characteristic, questions are defined and metrics answer these questions (*Goal Question Metric* (GQM) approach). Once we have determined the metrics that we want to use to assess the test artifact, we can analyze it by collecting and evaluating these metrics. The metrics on the one hand deliver values that allow an assessment and on the other hand, they deliver locations within the artifact that may need improvement. Based on this information, the test artifact can then be improved. For this purpose, we perform refactoring, i.e., we improve the quality characteristic under analysis without changing the semantics of the test specification.

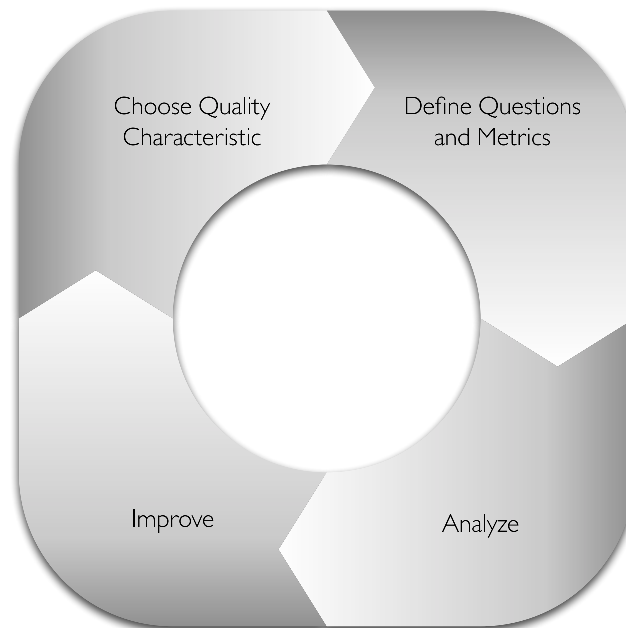


Figure 1.4.: The Analytical Quality Assurance Cycle

The concrete contributions of this thesis are the following:

- A quality model for test specifications which examines the different quality characteristics and subcharacteristics of test specifications. It presents the basis for this thesis by providing answers to the questions “What do we need to analyze?” and “What quality characteristics can we improve?”. The quality model enables a target-oriented choice of the aspects that should be assessed and provide an important building block for the application of the GQM approach.
- An exemplary instantiation of this quality model for quality characteristics of a TTCN-3 project. Here, we present how exactly the GQM is applied to specify questions and to determine metrics for a project using a concrete testing language. The instantiation also demonstrates that metrics can be language-specific—in our case, specific to the TTCN-3 language.
- A model-based analysis method for test specifications which provides an answer to the question how dynamic testing can be performed on tests in a feasible way. It constitutes the following:
  - A reverse engineering algorithm to construct *Extended Multi-Input/Output Transition System* (EMIOTS) models from high-level test specifications. The reverse engineering model allows a subsequent analysis to assess and locate quality characteristics.

 - A catalog of test specific, TTCN-3-specific, and generic properties that describe specific anomalies in test cases. These properties benefit from an analysis using the reverse engineered model or are not possible to analyze statically. The catalog is a guide and a starting point for test quality engineers to enable the design of their own custom properties.

- A method for the detection of test cases with inconsistent responses in a test suite. Inconsistent responses happen when two test cases are considered to have matching behavior up to a certain state. However, in this state, one test case expects different responses from the SUT than the other. In such occurrences, the test cases are considered to violate a response consistency criterion. The presented response consistency relation and the provided detection algorithm allow test quality engineers to detect such inconsistencies in test suites.

The case study in Section 5 presents how the approaches have been implemented in a prototype. This prototype is then applied to test cases of standardized test suites developed at ETSI to validate the contributions. The composite of the contributions provide a framework for the application of analytical quality assurance by providing the necessary methods to perform dynamic testing and inconsistency analyses that are novel to the quality engineering of test specifications.

1.5. Impact

The results of this dissertation have been peer-reviewed and published in several international workshop and conference proceedings as well as in various journals. In the following, we list workshop and conference publications with relation to the content in this dissertation:

- SAM 2006: *Refactoring and Metrics for TTCN-3 Test Suites.* Benjamin Zeiss, Helmut Neukirchen, Jens Grabowski, Dominic Evans, Paul Baker. LNCS 4320.
- TAIC PART 2006: *TRex – The Refactoring and Metrics Tool for TTCN-3 Test Specifications.* Paul Baker, Dominic Evans, Jens Grabowski, Helmut Neukirchen, Benjamin Zeiss.
- SE 2007: *Applying the ISO 9126 Quality Model to Test Specifications – Exemplified for TTCN-3 Test Specifications.* Benjamin Zeiss, Diana Vega, Ina Schieferdecker, Helmut Neukirchen, Jens Grabowski. LNI 105.
- SDL Forum 2007: *TTCN-3 Quality Engineering: Using Learning Techniques to Evaluate Metric Sets.* Edith Werner, Jens Grabowski, Helmut Neukirchen, Nils Röttger, Stephan Waack, Benjamin Zeiss. LNCS 4745.
- TESTCOM/FATES 2008: *Reverse-Engineering Test Behavior Models for the Analysis of Structural Anomalies* (Short Paper). Benjamin Zeiss, Jens Grabowski.

- SDL FORUM 2009: *Towards an Integrated Quality Assessment and Improvement Approach for UML Models.* Akthar Ali Jalbani, Jens Grabowski, Helmut Neukirchen, Benjamin Zeiss.
- TESTCOM/FATES 2009: *Analyzing Response Inconsistencies in Test Suites.* Benjamin Zeiss, Jens Grabowski. LNCS 5826.

The subsequent list presents the published journal and online journal articles with relation to in this dissertation:

- STTT Vol. 10(4): *An Approach to Quality Engineering of TTCN-3 Test Specifications.* Helmut Neukirchen, Benjamin Zeiss, Jens Grabowski. 2008.
- STVR Vol. 18(2): *Quality assurance for TTCN-3 test specifications.* Helmut Neukirchen, Benjamin Zeiss, Jens Grabowski, Paul Baker, Dominic Evans. 2008.
- OBJEKTspektrum Online Themenspezial Testing: *Systematische Qualitätssicherung für Testartefakte.* Jens Grabowski, Philip Makedonski, Thomas Rings, Benjamin Zeiss. 2009.

Furthermore, the author identified the topics and supervised two master theses, one bachelor thesis, and one project work with some relation to the overall topic of this thesis:

- Dennis Neumann: *Test Case Generation using Model Transformations.* Master Thesis. 2009.
- Lucas Schubert: *An Evaluation of Model Transformation Languages for UML Quality Engineering.* Master Thesis. 2010.
- Stefan Kirchner: *Documentation Generation for TTCN-3.* Bachelor Thesis. 2009.
- Dennis Neumann: *Entwurf und Weiterverarbeitung eines XML-Formates zur Speicherung von Transitionssystemen.* Project Work. 2008.

The tools and technologies created during the development of this thesis have been developed further (as the *t3q* and *t3d* tools) for use within the test standardization at ETSI and are actively used, for example, for checking the 3GPP LTE test suites during their development.

1.6. Thesis Structure

The structure of this thesis is as follows: in Chapter 2, we introduce the prerequisites of this thesis that are needed across all chapters. Chapter 2.1 introduces TTCN-3 [55], a test specification and implementation language standardized by ETSI. We use TTCN-3 for our practical discussions and experiments. In Chapter 2.2, we provide basic definitions and terms for the formal models that we use in the chapters on model-based analyses. Chapter 2.3 introduces temporal logic and model checking.

Chapter 3 discusses software quality models (Chapter 3.1). We discuss how these models can and must be adapted for the domain of test specifications and testing in general (Chapter 3.2) and present such an adaption in detail (Chapter 3.3). We show how to instantiate such a quality model by means of metrics, smells, static analysis, and dynamic analysis for the assessment and refactoring for the improvement (Chapter 3.4). Finally, we provide an exemplary instantiation of quality model characteristics for a project using TTCN-3 and discuss how to develop appropriate metrics for the discussed subcharacteristics (Chapter 3.5). The chapter concludes with a discussion of the related work on quality assurance approaches for test specifications and static analyses of general-purpose software and test specifications (Chapter 3.6).

The next chapter discusses the model-based analysis of test specifications (Chapter 4). We explain the terms and techniques involved in reverse engineering (Chapter 4.1) and present a formal model for representing test cases and test suites (Chapter 4.2). Based on this, we provide a reverse engineering method and algorithm that is applicable to abstract test specification (Chapter 4.3). Based on the reverse engineered model, we provide a catalog of generic applicable properties that can then be analyzed using this method (Chapter 4.4). Model-based analyses for test suites as a whole are described in Chapter 4.5 where we discuss a specific kind of consistency criterion between two test cases that we call response consistency. We conclude the chapter with a survey of related work (Chapter 4.6) on dynamic analyses for test specifications and dynamic analyses in general.

The practical application of analytical quality assurance to industrial-size test suites is demonstrated in Chapter 5. We first describe the implementation that we have developed for the static and dynamic quality assessment as well as for the improvement of TTCN-3 test specifications (Chapter 5.1). The actual application of the analysis software to TTCN-3 test specifications is presented in Chapter 5.2. We first measure quality attributes statically and associate automated refactorings to the assessment to improve the test suite. In a second experiment, we validate the feasibility of our dynamic analysis approach.

The last chapter (Chapter 6) concludes this thesis. We summarize our efforts and provide an outlook how our work can be refined further or what kinds of new research directions have emanated from our research results. Chapters 3 and 4 have been written to be self-contained, i.e., only the prerequisites chapter (Chapter 2) is a necessary dependency to understand the respective chapters. Chapter 5 builds on the results of both Chapter 3 and 4.

2. Prerequisites

In this chapter, we present the prerequisites of this thesis that will be used and referred to throughout this entire work. We introduce the core elements of the test specification and implementation language TTCN-3 (Section 2.1), formal models for describing test behavior (Section 2.2), and a short introduction to model checking (Section 2.3).

2.1. Testing and Test Control Notation 3 (TTCN-3)

The *Testing and Test Control Notation* (TTCN-3) [55, 67] is a testing language standardized by the *European Telecommunications Standards Institute* (ETSI). It is the successor of the *Tree and Tabular Combined Notation* (TTCN) which has been widely used for test suite standardization and testing—especially in the telecommunications sector. TTCN was originally a part of the ISO standard 9646 [86], the *Conformance Testing Methodology and Framework* (CTMF) for the *Open Systems Interconnection* (OSI) [85]. The strength of TTCN-3 is system testing, i.e., specification-based black-box testing, where an SUT is stimulated with input messages and the subsequent reactions are observed and assessed. Since its introduction, TTCN-3 has been adopted in a wide variety of new areas, such as the automotive domain, the avionics domain, or the health care sector. Its use has also been extended to different test types, such as performance testing [42].

The main presentation format for TTCN-3 is textual (the core notation) and has a lot of syntactical similarity to a general-purpose language like C or Java. However, unlike a general-purpose language, TTCN-3 is tailored for the purpose of test specification and retains a lot of concepts from TTCN while introducing new concepts as well. In the following, we will present the most important concepts of TTCN-3 along with small examples.

2.1.1. Distributed Testing

TTCN-3 introduces the concept of distributed testing as a standard language feature. Distributed testing means that the test behavior is distributed across several *Parallel Test Component*s (PTCs) that execute behavior concurrently. The PTCs may send messages among each other to synchronize their behavior or exchange data. Figure 2.1 illustrates a TTCN-3 test system architecture with multiple components, i.e., a *Main Test Component* (MTC) and multiple PTCs. The test components communicate with each other and the SUT via *ports*, where each port (illustrated by circles) of a test component has an input queue. By connecting or mapping ports among the test components or between the test components

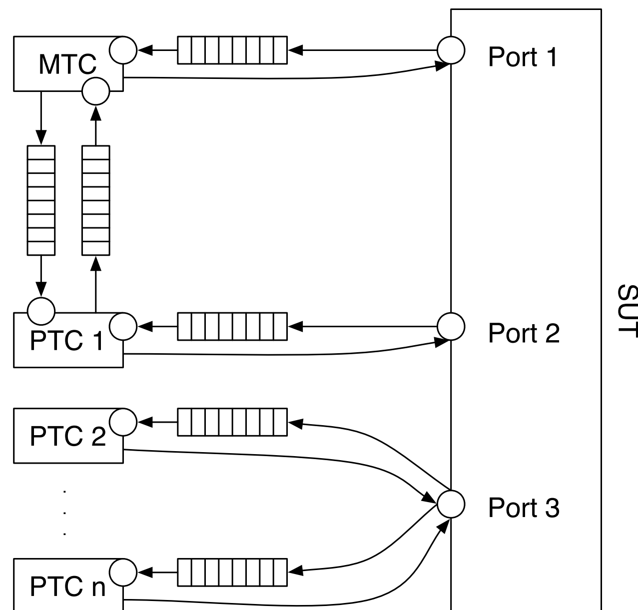


Figure 2.1.: TTCN-3 Test System Architecture

and the SUT, they are configured for communication. Due to the presence of queues, the communication paradigm of TTCN-3 is non-blocking, i.e., the sender does not wait for any confirmation from the sender and carries on directly. A test architecture frequently found in practice is the distribution of behavior corresponding to each port to a different test component. As mentioned before, the PTCs may communicate internally with each other or through the *Test System Interface* (TSI) with the SUT. The overall setup how the ports of the MTC, of the PTCs, and of the TSI are connected and mapped among each other is called *test configuration*.

2.1.2. Templates

Templates are the means to define and use test data in TTCN-3. The test data is defined according to specific predefined types for the templates. Templates do not only contain concrete values, but they may also utilize *matching operators*, which can be used to define a range or domain of test data. Such matching operators are, for example, optional values or wildcards. In addition, the type system is very comprehensive and allows a lot of restrictions, such as subtyping by range restriction. The actual test data that is received during a test execution is checked against a template to decide whether it matches. Templates that are used for sending data must be concrete and thus may not contain any special matching operators, such as wildcards. Listing 2.1 presents a simple template definition with matching operators along with the referenced type definition.

```
 1 module TemplateExample {
 2   type record Address {
 3     charstring street,
 4     integer zipCode,
 5     charstring city,
 6     charstring country
 7   }
 8
 9   template Address myAddress := {
10     street := "Goldschmidtstrasse 7",
11     zipCode := "37077",
12     city := "Goettingen",
13     country := "Germany"
14   }
15
16   template Address hamburgAddress := {
17     street := ?,
18     zipCode := ?,
19     city := "Hamburg",
20     country := "Germany"
21   }
22
23 }
```

Listing 2.1: Example: TTCN-3 Template

In lines 2–7, a type for the template is defined—an address record. Then, a concrete template is defined in lines 9–14. A matching template can be found in lines 16–21 that would match any message of type *Address* that have data values “Hamburg” and “Germany” in the city and country field respectively. The street and zipCode fields may have arbitrary values.

2.1.3. Behavior, Alternative Behavior, and Defaults

Behavior in TTCN-3 is specified by means of either the *testcase*, *function*, or *altstep* constructs. Testcases are compounds that specify the starting point of a test case behavior. Functions are subroutines used to perform a specific task that is to some degree independent of the code that calls the function. The altstep is a construct encapsulating *alt-statements*. Alt-statements can be regarded as a specialized switch statement in which behavioral decisions are taken based on when and whether certain messages arrive at test component ports. Before such an alt-statement is evaluated, a snapshot of the input queues is taken in order to capture one specific state of the queues in an evaluation step. Such alternative behavior of an alt-statement can be encapsulated in a specialized kind of subroutine, called an *altstep*. Altsteps can be explicitly referenced (just like a function) or they can be activated as so called default behavior. Defaults specify an altstep that is dynamically attached to the end of every alt-statement that is evaluated during the test execution. This way, common behavior, such as error catching behavior (such as unexpected incoming messages or timer timeouts), can be implicitly called without cluttering the test code. On the downside, due to

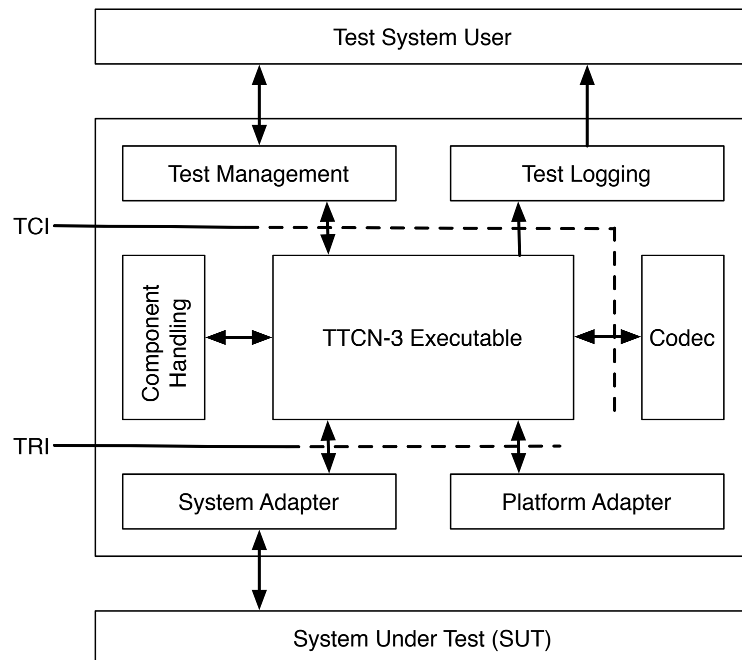


Figure 2.2.: TTCN-3 System Structure

its implicit nature, it may not always be clear to test engineers who are unfamiliar with the test code when exactly such a default is attached to an alt-statement.

2.1.4. Verdict Mechanism

TTCN-3 features a built-in mechanism for handling test verdicts. Each test component maintains its own local verdict. Additionally, there is a global test case verdict that is updated when test components terminate their execution. The global verdict is returned when a test case terminates its execution whereas the test component verdicts can be accessed directly within the TTCN-3 behavior. Valid verdicts in TTCN-3 are *none*, *pass*, *fail*, and *inconclusive*. In addition, TTCN-3 provides a set of rules on how verdicts can or cannot be overwritten. For example, a fail verdict cannot be overwritten or an inconclusive verdict can only be overwritten by a fail verdict.

2.1.5. Abstraction

A noteworthy characteristic of TTCN-3 is its architecture and how it promotes abstraction. Unlike generic scripting languages that can be used for testing, TTCN-3 is built upon a system that provides standardized services and capabilities that allow the adaption of the rather high-level test specifications to test implementations that can be executed. Abstract

test specifications in TTCN-3 can be compiled, but they are not executable by themselves. Instead, information needs to be added that allows an execution of the test specifications against a real-world SUT. Thus, we have a layered architecture where details are pushed into lower-level layers to keep higher-level layers abstract.

The interfaces for the parts that are required for an executable test specification are defined in the *Test Runtime Interface* (TRI) [57] and the *Test Control Interface* (TCI) [58]. With these interfaces, we can realize a test adapter that is able to encode and decode abstract messages into messages that are defined in the protocol under test. Furthermore, the adapter specifies how ports are mapped, how timers are realized, and similar.

Figure 2.2 illustrates the structure of a TTCN-3 test system. The TRI is more concerned with operations towards the SUT and platform adapters. The TCI focuses on test management, logging, and the codec.

2.1.6. Example

In the following, we present an abstract and simple, but complete example of TTCN-3 behavior. Listing 2.2 illustrates this example.

```
1  module testConfigEample {
2      type port myPort message {
3          inout integer
4      }
5      type component myComponent {
6          port myPort p;
7      }
8
9      template integer myMessage := 1;
10     template integer expectedMessage := 2;
11
12     altstep catchUnexpected() runs on myComponent {
13         [] p.receive {
14             stop;
15         }
16     }
17
18     testcase test() runs on myComponent {
19         var default myDefault := activate(catchUnexpected());
20         p.send(myMessage);
21         alt {
22             [] p.receive(expectedMessage) {
23                 setverdict(pass);
24             }
25         }
26         deactivate(myDefault);
27     }
28 }
```

Listing 2.2: Example: TTCN-3 Code

In the component definition (Lines 5 – 7), a port is defined. The interface definition for the test component and the TSI coincide. Therefore, the test case definition in Line 18

references only one component definition. In Line 19, a default is activated that will be dynamically attached to alt-statements from there on. A stimulus is sent to the SUT in Line 20. As a reaction to the stimulus there is an alt-statement (Lines 21 – 25) that handles expected and unexpected reactions to the stimulus. If the expected message arrives (Line 22), the verdict is set to *pass*. The other case is handled by the default altstep (Lines 12 – 16). In this default, the testcase is stopped when any other message than the expected message arrives. Note: in this case, no verdict is set. Finally, the default is deactivated in Line 26.

2.2. Formal Notations for Transition Systems

Model-based analysis of test specifications requires an appropriate formal model definition that is suitable for representing test specifications and capable to model those properties which are subject of the analysis. On the other hand, the formal model must be sufficiently simple. Otherwise, the definition of model analyses can easily become very complex. A model formalism often used for the representation of reactive systems is the *Labeled Transition System* (LTS). It forms the foundation for formal languages like the *Calculus of Communicating Systems* (CCS) [104] or *Communicating Sequential Processes* (CSP) [77]. It is generic enough to describe any kind of common existing languages while still being reasonably simple.

In the following, we give a few essential definitions that form the foundations for the model-based analyses in this thesis.

2.2.1. Formal Models

We start by giving a definition for the LTS, with actions partitioned into inputs and outputs (I/O). The I/O partitions give an additional context to each message—the information could theoretically also be encoded as information in the action itself. In addition, we describe unobservable actions by their own partition.

Definition 2.1 (Labeled Transition System (LTS) with I/O Partitions) *An LTS M is defined by the tuple (S, A, λ, s_0) where*

- *S is a finite and non-empty set of states,*
- *A is a set of actions that is partitioned into the sets of input actions A_I, the set of output actions A_O, and the set of unobservable internal actions A_N, i.e., $A = A_I \cup A_O \cup A_N, A_I \cap A_O \cap A_N = \emptyset$,*
- *λ is a transition relation with $\lambda \subseteq S \times A \times S$,*
- *s_0 is the initial state.*

A transition from the set λ is either written as triple (s, a, s') or as $s \xrightarrow{a} s'$.

We also refer to the tuple elements of the model by using them as index of M, e.g., M_S refers to the set of states in M. The elements of each set may have an upper index to refer to the model they belong to, for example, s_i^M refers to a state $s_i \in M_S$. To ease the distinction between input actions and output actions, we also use the notation $?a$ if $a \in A_I$ and $!a$ if $a \in A_O$. We use the notation $p!a$ or $p?a$ if a message a sent through a channel p or received through a channel p respectively[1].

Since channels are not explicitly a part of the LTS model, the channels can be interpreted as a label prefix. Due to our partitioning in inputs A_I, outputs A_O, and internal actions A_N, we are not in need of a special τ symbol that avoids multi-way synchronization—actions in A_N are never synchronized between communicating models (see Section 2.2.2). However, we use the τ symbol to refer to unnamed internal transitions.

Another model that we use is the *Extended Multi-Input/Output Transition System* (EMIOTS). We derive this definition from our LTS definition with input and output partitions and as well add partitions for channels, variables, and guards. By adding channels, a message is not received and sent globally to the system, but via specified channels.

Definition 2.2 (Extended Multi-Input/Output Transition System (EMIOTS)) *An EMIOTS M is defined by the tuple* (S,V,P,A,λ,G,s_0) *where*

- *S is a finite and non-empty set of states,*
- *V is an n-dimensional linear space $v_1 \times v_2 \times \ldots \times v_n$ representing global variables,*
- *P represents the set of channels,*
- *A is a set of actions that is partitioned into the sets of input actions A_I, the set of output actions A_O, the set of unobservable internal actions A_U that manipulate variables $V \rightarrow V$, and the set of unobservable internal actions A_N, i.e., $A = A_I \cup A_O \cup A_U \cup A_N, A_I \cap A_O \cap A_U \cap A_N = \emptyset$. The set of input actions A_I and the set of output actions A_O are each partitioned further into a finite number of disjoint non-empty sets A_I^p and A_O^p where the upper index $p \in P$ represents a channel of the communicating systems,*
- *G is a set of guard predicates g_i over the set of variables $g_i : V \rightarrow \{0,1\}$,*
- *λ is a transition relation with $\lambda \subseteq S \times G \times A \times S$,*
- *s_0 is the initial state.*

A transition from the set λ is either written as quadruple (s,g,a,s'), or as a triple (s,a,s'). In the latter case, the guard is disregarded, i.e., any specified predicate matches. The notation $s \xrightarrow{g/a} s'$ is equivalent to (s,g,a,s') and likewise $s \xrightarrow{a} s'$ again disregards the guard predicate.

[1] In this thesis, we discuss properties of test cases and test suites rather than properties of the tested system. Therefore, inputs and outputs take the view of the test case for a more intuitive understanding from its perspective. This means that inputs in our models are inputs to the test case, i.e., responses from the system, whereas outputs are outputs to the system, i.e., the stimuli.

An EMIOTS holds an extra space of variables, where a $v \in V$ is described by the n-tuple $v = (v_1, v_2, \ldots, v_n)$ denoting the variables in M. Only actions from the set A_U can manipulate the set of variables and the values of any v_i. Due to this fact, a state can become ambiguous in the sense that a variable may take on different values within a state. A transition is only applicable when the guard predicate evaluates to true. An empty predicate is denoted by "$-$", which always evaluates to true. Intuitively, the underlying structure of the EMIOTS depicts the behavioral control-flow, while the variables and the variable manipulation depict the data-flow. As a result, the EMIOTS model can have a more compact form than its corresponding LTS representation.

We distinguish between the terms *port* and *channel*. With the term port, we describe an access point of a communicating system. With the term channel, we describe an established connection between behavioral entities through which messages are received and sent. A channel has two ends, a sender and a receiver, whereas a port describes only one end with no established connection in between.

In the following, we provide various definitions for terms based on these models. With the introduced notations, the definitions are applicable to both the LTS and EMIOTS models. As the LTS model does not include guards, the provided guards in the following definitions simply always evaluate to 1.

Definition 2.3 (Path) *A path $s_i \xrightarrow{\sigma} s_n$ in M is a finite and non-empty sequence $\langle s_i, g_i, a_i, s_{i+1}, g_{i+1}, a_{i+1}, \ldots, a_{n-1}, s_n \rangle$ with $i, n \in \mathbb{N}$ such that the transitions $s_j \xrightarrow{g_j/a_j} s_{j+1}, j \in \mathbb{N}, i \leq j < n$ exist in λ.*

Definition 2.4 (Traces) *A trace σ in M is a finite sequence $\langle a_i, a_{i+1}, \ldots, a_{n-1} \rangle$ such that the sequence $\langle s_i, g_i, a_i, s_{i+1}, g_{i+1}, a_{i+1}, \ldots, a_{n-1}, s_n \rangle$ is a path in M. We denote the set of all traces over a set of actions A with A^*. We concatenate actions to denote action sequences using the $\cdot$ sign, e.g., $?a \cdot !b \cdot ?c \cdot !d$ would denote a sequence of actions that is read from left to right.*

The notation $s \xrightarrow{a_1 \cdot a_2 \cdot \ldots \cdot a_n} t$ means that transitions $s \xrightarrow{a_1} s' \xrightarrow{a_2} \ldots \xrightarrow{a_n} t$ exist. We write $s \xrightarrow{a_1 \cdot a_2 \ldots \cdot a_n}$ to denote the set of states t with $s \xrightarrow{a_1 \cdot a_2 \ldots \cdot a_n} t$.

With the double arrow, we denote paths that skip unobservable actions in $A_U \cup A_N$, i.e., if we have a path $s \xrightarrow{a_1 \cdot a_2 \cdot a_3 \cdot a_4} s'$ where $a_1, a_4 \in A_I \cup A_O$ and $a_2, a_3 \in A_U \cup A_N$, we may write $s \xRightarrow{a_1 \cdot a_4} s'$ for the abstracted sequence of observable actions. We write $s \xRightarrow{\sigma}$ iff there exists a state t with $s \xRightarrow{\sigma} t$.

Furthermore, with $traces(M)$ we denote the set of all traces that can be produced in model M from the start state s_0, i.e., $traces(M) := \{\sigma \in A^* | M \xRightarrow{\sigma}\}$. Here, M refers to the initial state s_0 of M.

Definition 2.5 (Enabled Actions) *The set of enabled actions of a state s is defined as $enabled(s) := \{a | \exists s' \in S : (s, g, a, s') \in \lambda\}$, i.e., a state s is enabled if there exists a state s' with a transition (s, g, a, s') and the set $enabled(s)$ is the set of available actions from this enabled state. A state s is called a* ***deadlock state*** *if $enabled(s) = \emptyset$.*

Definition 2.6 (Determinism) *A model M is deterministic if for all paths $s \xrightarrow{\sigma} t$ and $s \xrightarrow{\sigma} t'$ in M $t = t'$ is implied.*

Definition 2.7 (Relabeling Operator) *The operator $[f]$ denotes the relabeling operator where the notation $M[f]$ refers to a model M where actions are relabeled by a function $f : A \rightarrow A$. The notation $M[a_1/a'_1, a_2/a'_2, \ldots, a_n/a'_n]$ refers to a model M where a_i is relabeled to a'_i for $a \leq i \leq n$.*

2.2.2. Parallel Composition

To depict the composite behavior of multiple sequential test system models that run in parallel, we define composition operators that realize parallel behavior by means of interleaving. This model imposes that the order of concurrent actions is arbitrary and thus the interleaving results in all possible action orderings.

The synchronous parallel composition operator defines the fundamental set of rules to interleave models. It interleaves the models M with N in such way that shared actions which exist in M_A as well as N_A must always be executed at the same time. All kinds of compositions, also of non-blocking queued systems, can be modeled with the synchronous composition operator. For example, the synchronous composition of a model with a corresponding queue yields a non-blocking system. We will refine the synchronous composition operator for queued systems though to keep the notations in a compact form. Whether the actions are inputs, outputs, or internal actions is disregarded here. The following definitions define parallel composition for EMIOTS models. Just like with the previous definitions, for LTS models, guards evaluate to 1 and variables and guards are dropped from the tuple definition.

Definition 2.8 (Synchronous Parallel Composition Operator) *Given two models M and N, the synchronous parallel composition $P = M||N$ is defined as follows:*

- $P_S = M_S \times N_S$.
- $P_V = M_V \cup N_V$.
- $P_A = M_A \cup N_A$.
- $P_G = M_G \cup N_G$.
- P_λ *is defined by the following inference rules:*
  - $(s,t) \xrightarrow{g/a} (s',t) \in P_\lambda$ *if* $s \xrightarrow{g/a} s' \in M_\lambda$ *and* $a \in M_A \backslash N_A$,
  - $(s,t) \xrightarrow{g/a} (s,t') \in P_\lambda$ *if* $t \xrightarrow{g/a} t' \in N_\lambda$ *and* $a \in N_A \backslash M_A$,
  - $(s,t) \xrightarrow{g \wedge g'/a} (s',t') \in P_\lambda$ *if* $s \xrightarrow{g/a} s' \in M_\lambda$ *and* $t \xrightarrow{g'/a} t' \in N_\lambda$ *and* $a \in N_A \cap M_A$.
- $P_{s_0} = (M_{s_0}, N_{s_0})$.

The third inference rule requires the logical conjunction of the guard operators of both in the composed system to enforce that both predicates from M and N evaluate to true. Finally, we need a composition operator that composes the behavior of two models by synchronizing inputs and outputs. However, with this operator, we need to respect the variable, channel, and guard sets as well.

Definition 2.9 (Multi-Input/Output Parallel Composition Operator) *Given two models M and N, the multi-input/out parallel composition $P = M||_M N$ is defined as follows:*

- $P_S = M_S \times N_S$.
- $P_V = M_V \cup N_V$.
- $P_A = M_A \cup N_A$.
- $P_P = M_P \cup N_P$.
- $P_G = M_G \cup N_G$.
- P_λ *is defined by the following inference rules:*
  - $(s,t) \xrightarrow{g/a} (s',t) \in P_\lambda$ *if* $s \xrightarrow{g/a} s' \in M_\lambda$ *and* $a \in M_A \backslash N_A$,
  - $(s,t) \xrightarrow{g/a} (s,t') \in P_\lambda$ *if* $t \xrightarrow{g/a} t' \in N_\lambda$ *and* $a \in N_A \backslash M_A$,
  - $(s,t) \xrightarrow{g \wedge g'/\tau} (s',t') \in P_\lambda$ *if* $s \xrightarrow{g/a} s' \in M_\lambda, a \in M_{A^i_O}$ *and* $t \xrightarrow{g'/a} t' \in N_\lambda, a \in N_{A^i_I}$.
- $P_{s_0} = (M_{s_0}, N_{s_0})$.

2.2.3. Queues

In practice, we often deal with systems that do not block when an output is sent. That way, the sending system does not have to wait for a confirmation from the receiver, but it can directly proceed with its behavior. We call this requirement for the receiver *input-enabledness* [95, 141].

Definition 2.10 (Input-Enabled) *A model M is input-enabled if all input actions are enabled in all states and thus inputs are non-blocking. More formally: $\forall s \in S : enabled(s) \supseteq A_I$.*

There are various different ways to model the input-enabledness requirement. One possibility is to model all possible inputs as self-loops in each state, i.e., the state does not change and the input is effectively discarded. In this case, the behavior does not advance. A second possibility is that undefined inputs lead to an error state where behavior halts. Another common solution is the usage of input queues. In the following we present input queues in more detail.

Figure 2.3 illustrates a simple case of a queued test system. The test system receives incoming messages on a single port by its input queue which is represented by a separate

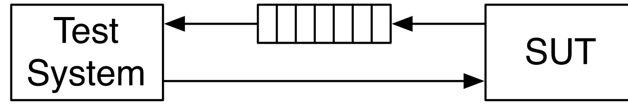


Figure 2.3.: Test System Input Queue

transition system. The input queue must always accept messages from the SUT and forward them to the test system in *First In, First Out* (FIFO) order. In our case, we focus on the behavior of the test system and thus disregard whether the SUT is queued or not. Implicitly, we assume that the SUT will always accept outputs from the test system. A technique to model this behavior in the test system is to create the queue as a separate model and then apply parallel composition to interleave the queue model and the test system model. A general problem is the fact that unbounded queues have infinite behavior. Hence, it is common practice to bound the queue in order to have a finite number of states. A behavioral model composed with a bounded queue model, however, still grows exponentially with its queue length.

The construction principle of the queue model is as follows: a queue of length one is modeled. This model is characterized by a start state and m number of states, where m also denotes the number of messages that can be accepted by the queue. The start state is then connected by an input transition to each state and each of these states is connected with an output transition to the start state. This initial queue, we call it BUF (see Definition 2.11), is then recursively interleaved with further copies of this queue, where the output actions of the recursive result are synchronized with input actions of BUF and then hidden in the result (or more compact: they are composed with the queue composition operator of Definition 2.12). Each subsequent interleaving extends the length of this queue by one. Figure 2.4 illustrates a queue of length two that receives and forwards messages r and s. In the illustration, we have merged states that are only reachable from a previous state by a single τ transition.

Definition 2.11 (Queue) *For the set actions A that shall be receivable on the port to be queued, we create an intermediate model called* BUF. BUF_S *consists of* $|A|+1$ *states, i.e., one state for each possible output message* $s_1, s_2, \ldots, s_{|A|} \in BUF_S$ *and a start state* $s_0 \in BUF_S$. *For each state (and message)* $s_i, 0 < i \leq |A|$, *there is one input action* $?a_i \in BUF_{A_I}$ *and a corresponding output action* $!a_i \in BUF_{A_O}$ *with the transitions* $s_0 \xrightarrow{?a_i} s_i \in BUF_\lambda$ *and* $s_i \xrightarrow{!a_i} s_0 \in BUF_\lambda$. *Based on* BUF, *the queue* Q *is recursively defined as follows:*

$$Q_1 = BUF \tag{2.2.1}$$
$$Q_n = (BUF \quad \|_Q \quad Q_{n-1}) \tag{2.2.2}$$

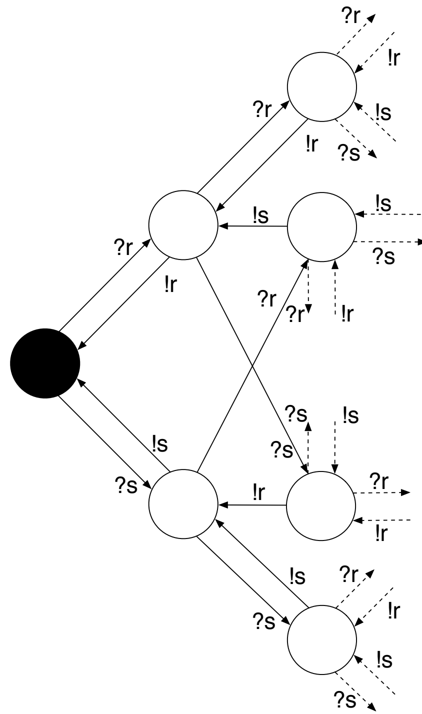


Figure 2.4.: Queue of Length 2 Accepting Messages *r* and *s*

Note that the number of actions $|Q_A|$ does not change with growing queue length and corresponds to the number of initial action $|A|$ from which the queue is constructed. The recursive definition as given has infinite length and behavior. A bounded queue stops the recursion at a predefined point. The queue length then corresponds to recursion level n. Also, the definition states that the queue accepts a set of actions A. Having an infinite set A would lead to infinite behavior as well—independently from the queue length.

In order to define queued behavior of a model, we define a special queue composition operator that synchronizes output actions with their corresponding queue input actions. This operation can also be accomplished by using the already defined synchronous parallel composition operator (Definition 2.8). However, relabeling operations (Definition 2.7) would be necessary to match corresponding input and output actions. As the notation with relabeling operations becomes verbose very quickly, we define a custom composition operator that fulfills this purpose and that keeps the overall notation more compact and less complex.

Definition 2.12 (Queue Composition Operator) *Given two models M and Q, with Q denoting a queue, the queue composition $P = M\|_Q Q$ is defined as follows:*

- $P_S = M_S \times Q_S$.
- $P_V = M_V$.
- $P_A = M_A \cup Q_A$.
- $P_G = M_G$.
- P_λ *is defined by the following inference rules:*
  - $(s,t) \xrightarrow{g/a} (s',t) \in P_\lambda$ *if* $s \xrightarrow{g/a} s' \in M_\lambda$ *and* $a \in M_A \backslash Q_A$,
  - $(s,t) \xrightarrow{g/a} (s,t') \in P_\lambda$ *if* $t \xrightarrow{g/a} t' \in Q_\lambda$ *and* $a \in Q_A \backslash M_A$,
  - $(s,t) \xrightarrow{g/\tau} (s',t') \in P_\lambda$ *if* $s \xrightarrow{g/a} s' \in M_\lambda, a \in M_{A_I}$ *and* $t \xrightarrow{g'/a} t' \in Q_\lambda, a \in Q_{A_O}$.
- $P_{s_0} = (M_{s_0}, N_{s_0})$.

A queue does not contain any variables. Consequently, we ignore the set of variables from the queue N in the composition. Also, queues should not have any guards. Therefore, we ignore the g' guard in the third inference rule.

To finally create an non-blocking model M' of a test system model M using a queue Q that has been constructed from the set of actions M_{A_I}, these two models have to be interleaved: $M' = M \quad \|_Q \quad Q$.

2.2.4. Implementation and Equivalence Relations

In this subsection, we introduce equivalence and implementation relations needed for the discussion in Chapter 4.5. Implementation relations are preorders described in the context of testing: they relate models of implementations and specifications. An implementation I conforms to a specification S only if the implementation relation relates I to S. Such relations are used to describe the generation of test cases.

Probably the most straightforward implementation relation is *trace preorder* $\leq_{tr}$ [17] which relates two labeled transition systems. Trace preorder requires the inclusion of trace sets and is formally defined as follows:

Definition 2.13 (Trace Preorder) *Let I, S be LTSs having a common set of actions A. Then,* $I \leq_{tr} S$ *if* $traces(I) \subseteq traces(S)$.

Another well-known implementation relation is the *ioco* relation [139]. In this relation, an implementation I is conformant to a specification S when a subset of the outputs (i.e., the responses of I and S) that can be produced in a state after each possible trace in the specification is present in the implementation as well.

Equivalence relations on the other hand are relations specify a partitioning that decides when two models are considered to be equal, i.e., they are considered equal when they belong to the same partition. In terms of models, an equivalence relation describes when we consider models to be equal. Especially when comparing behavior, defining such a relation is not always trivial as there are often many ways to express semantically identical behavior.

The most intuitive way to deal with this problem is to require trace equivalence, i.e., $traces(I) = traces(S)$. However, a restriction of trace equivalence is that it is not able to deal with non-determinism. For that purpose, the bisimulation relation [104] has been defined which is able to deal with this limitation. We differentiate between strong and weak bisimulation. Strong bismulation relates all transitions in a transition system, whereas the weak bisimulation relation relates only observable actions. We provide a definition of weak bisimulation defined for two separate transition systems (whereas the usual definition is given over a single transition system).

Definition 2.14 (Weak Bisimulation) *Given two LTSs T_1 and T_2, a binary relation $R \subseteq T_{1_S} \times T_{2_S}$ is a weak bisimulation iff the following conditions hold for every $(s,t) \in R$ and an action $a \in (T_{1_A} \cup T_{2_A})$:*

- *$s \xrightarrow{a} s' \in T_{1_\lambda}$ implies that there is a t' in T_{2_S} such that $t \overset{a}{\Rightarrow} t' \in T_{2_\lambda}$ and $(s',t') \in R$.*
- *Symmetrically: $t \xrightarrow{a} t' \in T_{2_\lambda}$ implies that there is an s' in T_{1_S} such that $s \overset{a}{\Rightarrow} s' \in T_{1_\lambda}$ and $(s',t') \in R$.*

We call two states s and t weakly bisimilar or $s \approx t$ iff $(s,t) \in R$. Similarly, we call two LTSs T_1 and T_2 weakly bisimilar, or $T_1 \approx T_2$, iff for every $s \in T_{1_S}$, there exists a $t \in T_{2_S}$ such that $s \approx t$ and for every $t \in T_{2_s}$, there exists an $s \in T_{1_S}$ such that $s \approx t$.

2.3. Model Checking

Model checking is a method pioneered by Clarke, Emerson, Queille, and Sifakis [34, 35, 36, 47, 122], in which a model is automatically verified against a specification. The model M is a simplified and possibly partial behavioral description and the specification is a formula ϕ phrased in a temporal logic. Model checking then verifies whether for a given model M and a logical property ϕ, M satisfies ϕ, i.e., $M \models \phi$. The model is described in a language that is usually formally similar to an *Extended Finite State Machine* (EFSM). The formula to be checked is described in a temporal logic like *Linear Temporal Logic* (LTL) [121] or *Computation Tree Logic* (CTL) (introduced in [35] as well).

Model checking is faced with a problem called the state *space explosion problem*. This problem is especially evident with largely independent parallel behavior where behavior can be interleaved in a huge number of combinatorial variations. The more complex a state

machine gets in number of states and transitions and the more state machines are interleaved in parallel, the more transitions and states exist in the composite machine combinatorially. If this problem was left unconsidered, model checking would not be computationally applicable in practice. Modern model checking methods mitigate this problem for many cases: symbolic algorithms avoid to build the complete interleaved graph for the overall state machine [99], partial-order reduction [66] reduces the number of possible interleavings by removing irrelevant permutations that do not affect the verification of the formula, or bounded search ensures that the model checking procedure will eventually terminate, albeit with a less strong validity about the result.

2.3.1. Temporal Logic

To verify that a formula ϕ does or does not hold for a model M, the formula is described in a temporal logic. LTL is a modal logic with operators allowing us to refer to the future. Here, time can be regarded as a sequence of states that will be visited in time and the sequence of states can be regarded as a computation path. When we apply model checking, the model checking method takes control over determining which paths are visited. In an LTL formula, we are working with atoms (or propositional variables) which are facts that may hold in a system. An atom a might represent the fact "resource is busy" while atom b might represent the fact "resource is free". What kind of atoms we use and what kind of atoms we define depends on our verification interest. Our propositional variables are connected with usual logic operators such as $\neg$ (not), $\vee$ (or), $\wedge$ (and), $\rightarrow$ (implies) and temporal operators: $\circ$ (for next), $\Box$ (for always), $\Diamond$ (for eventually), $\mathcal{U}$ (for until), $\mathcal{W}$ (for weak until), and $\mathcal{R}$ (for release). The $\circ$ and $\Box$ operators are unary, i.e., they have one operand, while $\Diamond$, $\mathcal{U}$, $\mathcal{W}$, and $\mathcal{R}$ are binary, i.e., they require two operands. The binding priorities are as follows: unary operators (i.e., $\neg$, $\circ$, $\Diamond$, and $\Box$) have that tightest binding, next in order are $\mathcal{U}$, $\mathcal{W}$, and $\mathcal{R}$ followed by $\vee$ and $\wedge$. The $\rightarrow$ operator has the weakest binding. A simple example for a LTL formula would be:

$$\phi := \Diamond a \wedge \Box b \rightarrow \Box c.$$

In natural language, this formula would roughly translate into: if in some state a becomes true and b is always true implies that c is always true in all states. Logically, this means that the formula is always true except for the case when the statement "in some state a becomes true and b is always true" is true and the statement "c is always true in all states" is false.

There are various other temporal logics other than LTL. Also widely used is CTL. Even though CTL and LTL look roughly similar, the logics are incompatible to each other. Some formulas can be expressed in CTL that cannot be expressed in LTL and vice-versa. Both logics have strengths and weaknesses and there is essentially no correct choice for one of them as a formalism as long as the differences do not limit the expressiveness that the author needs. In the remainder of this thesis, we will use LTL as temporal logic.

2.3.2. Properties

With the term *properties*, we mean those properties of a software system that we intend to check with our analysis. We differentiate between *safety properties* and *liveness properties*. A safety property expresses that an event never occurs under certain conditions or that something bad never happens. An example for a safety property is negative invariants. Imagine an ATM, then a safety invariant would be that money is never withdrawn if an incorrect PIN has been entered. Liveness properties on the other hand state that some event will ultimately occur under certain conditions, i.e., something good will happen eventually and if the event has not happened yet, it will happen in the future. For example, at the ATM, money will be handed out once a correct PIN has been entered. In that sense, liveness properties require some kind of progress.

2.3.3. Property Patterns

Likely reasons for the slow adoption of verification techniques involving temporal logic are simple barriers, such as a lack of good tool support, little expertise, or hard to understand training materials [126]. A lot has changed since Rosenblum's article was written over ten years ago, for example, training materials have improved drastically and several well-written books covering this subject have been published (e.g., [14, 15, 16, 79, 129]). However, there is still some truth to the article: while there are well performing tools, it is still not very easy to use them and experts in this area are still rare. In particular, formulas in temporal logics may be precise, but they can be relatively hard to understand for non-trivial cases. A very useful means to create formulas in temporal logics are specification patterns as presented by Dwyer, Avrunin, and Corbett [45].

A specification pattern is a high-level and formalism independent specification abstraction. This is useful as problems that regard the description of temporal logic formulas are recurring. Rather than getting lost in the expressiveness of the logic, the practitioner can look for the right kind of specification pattern and use the formula template provided by this pattern to express it. And indeed, we notice that the properties that we present in the subsequent sections are often repetitive in their nature and that they match to the specification patterns identified by Dwyer et al. This reduces the complexity of the formula descriptions to the extent that the practitioners have to understand the pattern and that we can simply refer to the specification pattern name for them to immediately understand what is happening in the formula.

Dwyer et al. describe the following specification patterns:

- **Absence (Occurrence):** A given event does not occur within a scope.
- **Existence (Occurrence):** A given event must occur within a scope.
- **Bounded Existence (Occurrence):** A given event must occur k times within a scope.
- **Universality (Occurrence):** A given event occurs throughout a scope.

- **Precedence (Order):** An event P must always be preceded by events $Q_1, Q_2, \dots, Q_n$.
- **Response (Order):** An event P must always be followed by an event Q within a scope.
- **Chain Precedence (Order):** A sequence of events $P_1, P_2, \dots, P_n$ must always be preceded by a sequence of events $Q_1, Q_2, \dots, Q_n$.
- **Chain Response (Order):** A sequence of events $P_1, P_2, \dots, P_n$ must always be followed by a sequence of events $Q_1, Q_2, \dots, Q_n$.

Here, all patterns are categorized into *Occurrence* patterns and *Order* patterns (denoted in parentheses). For each of these patterns, temporal logic formula templates (for various kinds of temporal logics, including LTL) are provided for a set of given parameters in a given scope. As an example, for the *Absence* pattern, they provide the following LTL formula templates for "P is false":

- **Globally:** $\Box\, (\neg P)$.
- **Before R:** $\Diamond\, R \rightarrow (\neg P\, \mathcal{U}\, R)$.
- **After Q:** $\Box\, (Q \rightarrow \Box(\neg P))$.
- **Between Q and R:** $\Box\, ((Q \wedge \neg R \wedge \Diamond\, R) \rightarrow (\neg P\, \mathcal{U}\, R))$.
- **After Q until R:** $\Box\, (Q \wedge \neg R \rightarrow (\Box\, \neg P \vee (\neg P\, \mathcal{U}\, R)))$.

Similarly, the other patterns provide similar temporal logic templates. The complete specification pattern catalog can be found on a website dedicated to this catalog [4].

2.3.4. Analysis Classification

There are controversial discussions as to whether formal methods, such as model checking, abstract interpretation, which are based on models of the software product under analysis are considered static or dynamic analyses. On the one hand, the analysis of formal models as in model checking does not involve the actual execution of the real software product. They deal with simplified models of the actual system. Also, the execution does not take place in the same way as it is the case with actual code: the execution is optimized towards the verification of specific properties, allowing the execution engine to optimize the execution and, for example, skip certain permutations of the parallel behavior that do not have any influence on the outcome of the property to be verified (partial order reduction [66]). On the other hand, the models are still derived from the software product and the model checker still in essence executes the model in some sense to check the properties—real compilers optimize the code and the execution as well. These formal methods are somewhere on the borderline between static and dynamic, and it is not easy to draw a clear line. In this thesis, we consider such formal methods to be part of the dynamic analysis.

3. A Quality Model for Test Specifications

Analytical quality assurance is reactive, i.e., software—in our case test specifications—is developed and its quality is evaluated and improved after it has been written or while it is being written. To perform analytical quality assurance, two central questions need to be answered: the first question is **what** exactly we want to assess, measure, and evaluate. Once we are clear with the subject of the analysis, we can deal with the question of **how** we actually assess, measure, evaluate, and possibly improve. This chapter deals with the question **what** needs to measured.

In the assessment and evaluation, we differentiate between the terms *validation* and *verification*[2]. The differentiation was provided by Boehm in 1979 [22] by the questions he phrased to which the respective actions should provide answers:

- Validation: “Are we building the right product?”.
- Verification: “Are we building the product right?”.

With other words, verification is the checking whether the subject conforms to its specifications, regulations and whether it realizes its functional and non-functional requirements correctly and well. Validation on the other hand is more general and is more concerned with the satisfaction and acceptance of the product by the customer. A quality model for test specifications can be regarded as a building block for the verification of quality requirements as it provides a model for the subdivision of quality requirements and the answer to the question **what** needs to be measured.

In the context of the subsequent discussion, we use the terms *adaptation* and *instantiation*[3]. Quality models for software products are abstract, i.e., they are designed to be applicable to any software product no matter what its domain is. In order to make it suitable and more concrete for a specific domain, such as the testing domain, a quality model needs to be adapted. In the adaptation, the described attributes of the quality model are reinterpreted for this specific domain and possibly changed. In addition, there may be cases where the quality attributes of the respective model do not apply. The term instantiation on the other hand describes how a quality model or an adapted quality model is used for the quality assessment. The instantiation takes place by the definition of metrics that support

[2]The term *verification* has a different meaning in the context of the formal methods field. We refer to Boehms definition of *verification* by default and will use the term *formal verification* otherwise.

[3]We use the term *instantiation* in two contexts. First, an abstract test specification can be instantiated to make it executable. Second, a quality model can be instantiated with metrics for the concrete assessment of a software product. In this chapter, we discuss the instantiation of a quality model.

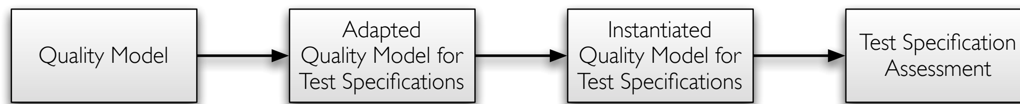


Figure 3.1.: Quality Model Adaptation Procedure

the assessment of the respective quality attributes. Here, the defined metrics are not general, but project-specific or even specific to the language that is used within a project. In this regard, we present in this chapter the adaptation of the model, describe the reinterpreted quality attributes of the adapted model, discuss how this adapted model can be instantiated in general terms, and finally provide a concrete example instantiation for a TTCN-3 project. Figure 3.1 illustrates the quality model adaption procedure as performed in this chapter.

This chapter is divided as follows: Section 3.1 discusses the currently available models for describing software quality, especially regarding the available standards. Section 3.2 discusses how we have adapted the ISO/IEC 9126 model for software quality for test specifications. Section 3.3 provides a detailed description for every characteristic and subcharacteristic of our adapted quality model for test specifications. The assessment and quality model instantiation methodology for the quality assessment is discussed in Section 3.4. Subsequently, we provide an example instantiation of the adapted quality model for test specification in Section 3.5. We conclude the chapter with a discussion on related work on quality assessment and measurement for test specifications, especially based on static analyses, in Section 3.6.

3.1. Software Quality Models

Quality models help to answer the first question, i.e., what the subject of our assessment is or what it should be. A variety of suggestions for software quality models have been made in the past. Well-known and established quality models are hierarchical models, so called *Factor-Criteria-Metrics* (FCM) models. The factors describe the relevant main quality characteristics that are subject of the evaluation. The criteria subdivide the characteristics into measureable attributes. Finally, metrics are mappings of a quantifiable quality attribute to a symbol or value on a specific measurement scale. Due to the dependency between characteristic and attributes and attributes and metrics, such models are regarded as hierarchical. However, depending on the concrete model, different characteristics may share the same quality attributes. Similarly, certain aspects of differing quality attributes may be measured with the same metrics. Therefore, the dependencies among factors, attributes, and metrics can have a structure of a net or graph rather than only the structure of a tree.

Prominent examples for these models are the McCall model [97] (that coined the term FCM), the Boehm model [23], and the ISO/IEC 9126 software quality model [88]. While all quality models are based on a hierarchy of different quality levels, their differences

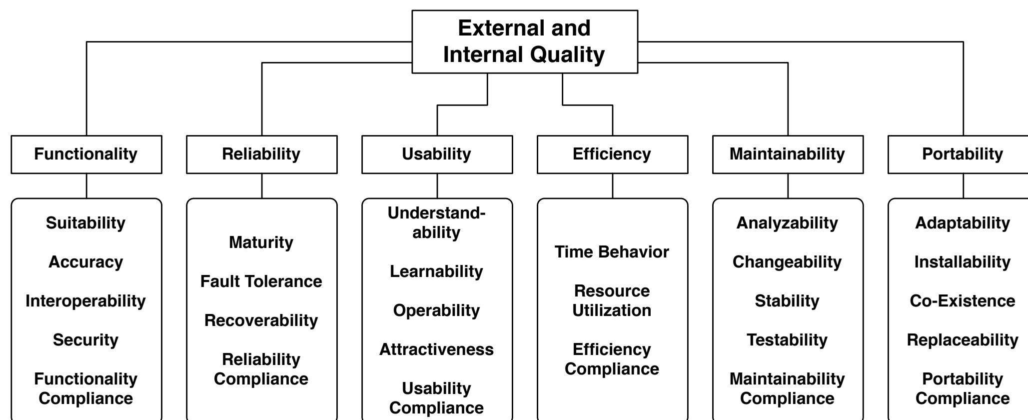


Figure 3.2.: ISO 9126 Quality Model

primarily lie in where they focus their measurements. For example, Boehm's quality model is based on a wider range of characteristics with a focus on maintainability when compared to McCall's model.

3.1.1. The ISO/IEC 9126 Standard

The ISO/IEC 9126 standard is based on the McCall and Boehm models and is very similar in its structure. The ISO/IEC 9126 standard describes a generic model for the internal and external quality of a software product and a model for the quality in use of a software product. *Internal quality* is obtained by reviews of specification documents, by checking models, by static analysis of source code, or by testing. *External quality* refers to properties of software when interacting with its environment. In contrast, *quality in use* refers to the quality perceived by an end user who executes a software product in a specific context. These product qualities at the different stages of the development are not completely independent, but influence each other. Thus, internal metrics may be used to predict the quality of the final product—even in early development stages.

For modeling internal quality and external quality, the ISO/IEC 9126 standard defines the same model. This generic quality model can then be instantiated as a model for internal quality or for external quality by using different sets of metrics that measure the respective characteristics and subcharacteristics. The model itself is based on the six characteristics *functionality*, *reliability*, *usability*, *efficiency*, *maintainability*, and *portability*. Figure 3.2 illustrates the hierarchical ISO/IEC 9126 quality model.

The model for *quality in use* is based on the characteristics *effectiveness*, *productivity*, *safety*, and *satisfaction* and does not elaborate on further subcharacteristics. Furthermore, the ISO/IEC 9126 standard defines metrics which are intended to be used to measure the

attributes of the respective quality models. However, the provided metrics are quite abstract and they cannot be applied without further refinement.

3.1.2. The FUPRS, FURPS+, and Dromey Quality Models

Other, not so well-known quality models are the FURPS/FURPS+ models [68] which originate from Robert Grady. In the FURPS model, characteristics are decomposed into two requirement categories: functional requirements (F) which are defined by input and expected output and non-functional requirements (URPS) that consist of usability, reliability, performance, and supportability. The main characteristics are very similar to the ISO/IEC 9126 model. FURPS+ additionally considers design requirements, implementation requirements, interface requirements, and physical requirements. Furthermore, Dromey [44] presents a quality model that recognizes that quality evaluation differs for each product. He attempts to connect software product properties with quality attributes and concentrates on the relationship between quality attributes and their subattributes and proposes a more dynamic idea for modeling the process.

3.1.3. Related Software Quality Standards

Other standards relevant to software quality, but not necessarily to FCM models are ISO/IEC 14598 (software product evaluation) [87], IEEE standard 1061-1998 [109], and ISO/IEC 25000 SQuaRE (software product quality requirements and evaluation) [90]. The ISO/IEC 14598 is more focused on the process of the evaluation rather than the product. The previously described ISO/IEC 9126 standard does not provide any description of how the evaluation should take place. The ISO/IEC 14598 standard fills this gap. The IEEE standard 1061-998 describes a methodology for establishing quality requirements and how the process is identified, implemented, analyzed, and validated as well as product software quality metrics. Finally, the ISO 25000 series will provide a new generation of standards for software product quality requirements and evaluation in the future. It is planned to include terms and definitions, reference models, and standards for requirements specification, planning, management as well as measurement and evaluation. The ISO/IEC 9126 and ISO/IEC 14598 are planned to be transitioned to the newer line of SQuaRE standards. This transitioning process is, however, not yet complete.

3.2. Adapting the ISO 9126 Quality Model for Test Specifications

The quality models presented in the previous section are models for assessing general-purpose software. Test specifications on the other hand can be regarded as software specifications for a specific domain—the testing domain. While generic models for general-purpose software might be applied to test specifications as well, an adapted quality model for this special domain is more natural to work with and it avoids different vocabulary or

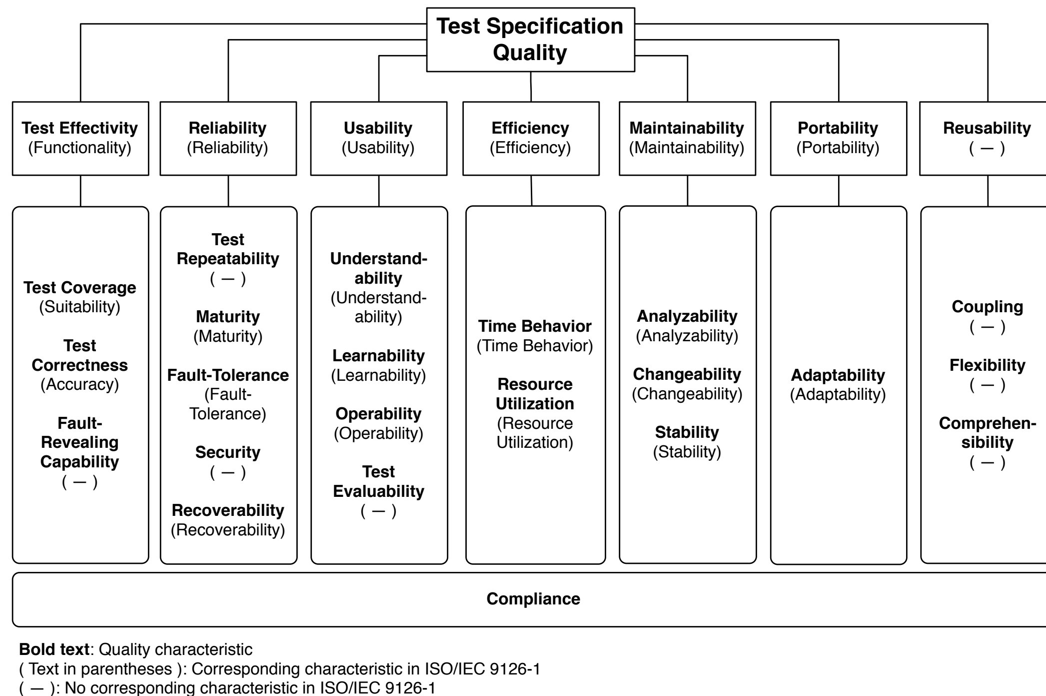


Figure 3.3.: ISO 9124 Adaptation for Test Specifications

interpretations of characteristics that might be ambiguous in this domain. In addition, we can remove characteristics that do not play a role in this domain. In fact, the ISO/IEC 9124 standard clearly states that “other ways of categorizing quality may be more appropriate in particular circumstances” (clause 5.4, paragraph 1). As the ISO/IEC 9124 model allows and encourages this kind of adaptation and due to the fact that this model is standardized, intuitive, and well-known, we have proposed to adapt this model for test specifications [156].

Figure 3.3 illustrates our test specification quality model. The model is divided into seven main characteristics: test effectivity, reliability, usability, efficiency, maintainability, portability, and reusability. Each main characteristic is structured into several subcharacteristics. To indicate the relationship of our test specification quality model (Figure 3.3) to the original ISO/IEC 9126 model (Figure 3.2), we provide the corresponding name of the ISO/IEC 9126 characteristics in parentheses. Test quality characteristics are printed in bold letters. Characteristics which have no corresponding aspect in ISO/IEC 9126, are denoted by the minus sign (–). The seven characteristics are explained in more detail in Section 3.3. The two most notable changes in the adapted quality model for test specifications are the new main characteristic *reusability* and the extraction of the *compliance* subcharacteristic to a more general all-encompassing subcharacteristic.

3.2.1. New Characteristics

The main characteristic *reusability* (right-hand side of Figure 3.3) is not covered in ISO/IEC 9126. We added it to our model as test specifications and parts of them are often reused for different kinds of testing, e.g., test cases and test data for system level testing may be reused for regression testing, performance testing, or testing different versions of the SUT. Thus, design for reusability is an important quality criterion for test specifications.

3.2.2. Changed Characteristics

In the original ISO/IEC 9126 model, each main characteristic contains a *compliance* subcharacteristic which denotes the degree to which the test specification adheres to potentially existing standards or conventions concerning this aspect. Since such conventions also exist for test specifications, they are also included in our model. However, as *compliance* is part of every quality characteristic, we feel that its inclusion in each set of subcharacteristics distracts from those subcharacteristics that are actually unique for each quality characteristic. As a result, we have moved it into a section of all-encompassing subcharacteristics. As *compliance* primarily alludes to conventions and standards that are mostly company or project-specific, we do not cover them any further.

Other changes include the renaming of the *functionality* characteristic to *test effectivity* and its subcharacteristics *suitability* and *accuracy* are characterized by the terms *test coverage* and *test correctness*. The reason for changing the term *functionality* is that test specifications do not have concrete functionality per se. Their functionality in essence is to find faults or problems in a piece of software. The difference lies in the way (or method) of how they attempt to fulfill their specific purpose (e.g., system testing or performance testing). Such methods can be, for example, structure-based or specification-based methods. However, the choice of methods in combination with their purposes and the degree to which a test realizes both methods are decisive for the usefulness and thus the effectiveness of a test. In that sense, the term *functionality* is misleading for test specifications, whereas the term *test effectivity* captures the quintessence of what can be interpreted from this category in the context of test specifications. Analogously, *test coverage* and *test correctness* are more suitable terms in the context of *test effectivity*.

3.2.3. Removed Characteristics

Various subcharacteristics have been removed from the adapted quality model as they are not applicable for test specifications. Their intent is covered by a different characteristic, or they have been moved to a different set of subcharacteristics. In the *functionality* characteristic, the *interoperability* and *security* subcharacteristics have been removed. The *interoperability* characteristic has been omitted from the test specification quality model as test specifications are too abstract for *interoperability* to play any role, i.e., interoperability

plays no role between multiple test specifications. This is not to be confused with interoperability testing between multiple systems that should be interoperable, for example, devices implementing the same specification, but which are built by different providers or hardware manufacturers. Within the context of tested systems, interoperability is an important aspect that is actively examined in research. The *security* aspect has been moved to the *reliability* characteristic, as it fits more appropriately to the term *reliability* than *test effectiveness* after renaming the main characteristic.

In the *usability* quality characteristic, the *attractiveness* subcharacteristic has been removed as it is not relevant for test specifications. *Attractiveness* may play a role for test execution environments and tools, but for plain test specifications, there is no user interface involved that could be attractive or not attractive to the end user.

In the *maintainability* quality characteristic, the *testability* subcharacteristic has been removed. The *testability* subcharacteristic describes the capability of a test specification to be validated after a modification. Design for testability, the main intent of this subcharacteristic in the original ISO/IEC 9126 quality model, is usually not relevant for the specification of tests as test specifications are tested technically differently than a general-purpose software product. Validating test specifications implies checking of either generic properties that need to be fulfilled or checking against a higher-level test purpose description. Both of these points are covered by the test correctness subcharacteristic. In order to allow such analyses, the used test specification language must be sufficiently formal. For example, an informal natural language test specification is harder to analyze than a formal test specification. The evaluation whether this kind of analysis is possible, is covered by the analyzability subcharacteristic. Hence, testability with its primary criterion "design for testability" does not play a role for test specifications.

Finally, in the *portability* quality characteristic, the subcharacteristic *installability*, *co-existence*, and *replaceability* have been removed. Test specifications are abstract. Therefore, *installability* (ease of installation in a specified environment), *co-existence* (with other test products in a common environment), and *replaceability* (capability of the product to be replaced by another one for the same purpose) are often too concrete for the concept of a test specification. Subcharacteristics that have been added are discussed in next section for each quality characteristic.

3.3. Quality Characteristics of the Quality Model for Test Specifications

In the following, we provide a discussion on the meaning of each characteristic and subcharacteristic of the quality model for test specifications.

3.3.1. Test Effectivity

The *test effectivity* characteristic describes the capability of a test specification to fulfill a given and specified test objective.

Test coverage constitutes a measure for test completeness and can be measured on various levels, e.g., the degree to which the test specification covers system requirements, the system specification, or test purpose descriptions. We roughly differentiate between specification-based, structure-based, and defect-based techniques for testing. Specification-based techniques are often based on system specifications or software requirements specifications. Here, coverage is determined, for example, by the degree to which requirements are covered by the test specification. Common techniques in this field are equivalence partitioning, boundary value analysis, cause-effect graphing, state transition tables (testing all valid states and transitions of a system modeled as an automaton), or pairwise testing (summaries for these methods are provided, for example, by Bath or Spillner [9, 134]). In structure-based testing, we deal with white-box criteria like statement testing, branch/decision testing, condition testing, or path testing. In defect-based testing, test specifications are developed against specific defect taxonomies (e.g., the defect taxonomy from Beizer [13]) that classify possible types of defects using knowledge about these specific defect categories. Each of these testing techniques specify criteria that test developers should aim to maximize. For example, in branch/decision testing, test developers should aim to test all branches of the tested behavior. In requirements based specification testing, the test developer should aim to cover all requirements of the requirements specification. The degree to which the criteria are fulfilled is determined by this subcharacteristic.

The *test correctness* characteristic denotes the correctness of the test specification with respect to the system specification or the test purposes. It describes states the degree to which a test specification realizes the test purposes and specifications that it is supposed to test. When test specifications are developed manually from such purposes or specifications, insufficient correctness may be a result of human mistakes in development, hard to understand specifications, or even ambiguous specifications. When test specifications are derived from specifications using model-based testing and automatic test generation, insufficient correctness may be the result from unsound test generation techniques or generally wrong assumptions. Furthermore, a test specification is correct only when it always returns correct test verdicts and when it has reachable end states.

The *fault-revealing capability* is a new item that has been added to the list of subcharacteristics. Obtaining a good coverage with a test suite having correct test cases does not make any statement about the capability of a test specification to actually reveal faults. The usage of cause-effect analysis [105] for test creation or mutation testing [40] help to assess the *fault-revealing capability* of test suites.

3.3.2. Reliability

The *reliability* characteristic describes the capability of a test specification to maintain a specific level of performance under different conditions. In this context, the word "performance" expresses the degree to which specific needs and requirements towards the test specification are satisfied. The *reliability* subcharacteristics *maturity*, *fault-tolerance*, and *recoverability* of ISO/IEC 9126 apply to test specifications as well. However, new subcharacteristics *test repeatability* and *security* have been added.

Test results should always be reproducible in subsequent test runs. Otherwise, debugging the SUT to locate a defect becomes hard or even impossible. Therefore, an important requirement for test specifications is *test repeatability*: a test specification should be designed to produce the same result, outcome, or verdict when executed multiple times against an SUT that is expected to be in the same state for each repeated test run. An important requirement towards this property of a test specification is that a test specification should be deterministic and, if possible, anticipate potential non-determinisms of the SUT. In that context, it is valuable to note that test specifications designed with parallel independently running test components that are not consequently synchronized always introduce some kind of non-determinism into tests due to different possible interleavings in the parallel behavior. Depending on how the SUT reacts to such different event orders introduced by parallel test components, such a test specification design (that is often chosen primarily for abstraction reasons) may already introduce problems regarding the test repeatability.

The *maturity* subcharacteristic denotes the capability of a test specification to deal with or avoid internal errors, e.g., a function returning out of bounds results. A test specification is mature when such errors on the one hand do not influence the execution of subsequent test cases and follow-up behavior, i.e., the test case is able to lead the SUT back into a known state when such an error occurs and the error or problem has not further influence on the remaining tests that follow. On the other hand, in order to achieve this level of stability, i.e., leading the SUT back into an expected state requires a very careful specification of the tests, a high degree of awareness for errors that could possibly occur, and appropriate verdicts need to be set accordingly. One aspect to achieve a mature test specification is to make it recoverable (see below).

Fault-tolerance describes the capability of a test specification to deal with or avoid external errors (e.g., from the SUT). For example, timers and corresponding timeouts are necessary to handle situations where the SUT does not respond. A fault-tolerant test case is not necessarily mature. For example, a test case might handle timeouts appropriately in order to terminate the test case. However, if the SUT is not reverted to the expected next state after the timeout occurs, subsequent test cases may be influenced due to the outcome.

The *security* subcharacteristic has been moved from the *functionality* characteristic. This subcharacteristic covers issues that are related to the protection of information or data that unauthorized persons may not be allowed to access. For example, the inclusion of plain-text passwords may play a role in the security aspect when test specifications with such data

are made publicly available or exchanged between development teams. Similarly, modern services often allow access to their interfaces only using cryptographically signed requests. The inclusion of private keys required to sign service requests is highly critical.

Finally, the *recoverability* subcharacteristic describes the capability to re-establish stable states when failures occur. A frequent architectural element used in test specifications to achieve this is the usage of preambles and postambles. They establish consistent and stable states before and after actual test behavior is executed. The maturity and recoverability aspect go hand in hand. Without good recoverability, a high level of maturity is not achievable. One aspect of maturity is the necessity that a test specification is recoverable also when something unexpected happens.

3.3.3. Usability

The *usability* attributes characterize the ease of actually instantiating or executing an instantiated test specification. This explicitly does not include usability in terms of difficulty to maintain or reuse parts of the test specification, which are covered by other characteristics. The original definition of usability described the “capability of the software product to be understood, learned, used and attractive to the user, when used under specified conditions”. However, a test specification is not used like an usual software product. Rather, it is an abstract document that describes formally how to test a software system. Only if instantiated, it can be executed against the system. Therefore, the following subcharacteristics relate to how well this abstract document can be handled by the developer and how well it is prepared for a test specification instantiation.

Understandability is important since the test users must be able to understand whether a test specification is suitable for their needs. Documentation and description of the overall purpose of the test specification are key factors—also for finding suitable test selections.

The *learnability* of a test specification pursues a similar goal as understandability. To properly use and instantiate a test suite, the user must understand how it is configured, what kind of parameters are involved, and how they affect test behavior. Proper documentation or usage of style guidelines have positive influence on this aspect.

A test specification has a poor *operability* if it, for example, lacks appropriate default values, or a lot of external, i.e., non-automatable actions are required in the test execution. Such factors make it hard to setup a test suite for execution or they make execution time consuming due to a limited degree of automation. Note that the operability aspect (as well as all other descriptions for this quality model) still regard the test specification as an abstract specification. In particular, this means that possible aspects in the sense of controlling an actual test execution do not play a role. Such aspects would refer to the test and test tool implementations that are not subject of the examination in the quality model for test specifications.

A new test-specific subcharacteristic in *usability* is *test evaluability*. The test specification must make sure that the provided test results are detailed enough for a thorough analysis of

an instantiated test specification. An important factor is the degree of detail richness in the specification of test logging. A test case that, for example, only specifies the report of a verdict answers the question of **what** happened, but it does not answer the question **why** it happened. The test evaluability subcharacteristics pay respect to the **why** question.

3.3.4. Efficiency

The *efficiency* characteristic relates to the capability of a test specification to provide acceptable performance in terms of speed and resource usage. The ISO/IEC 9126 subcharacteristics *time behavior* and *resource utilization* apply without any considerable change in their interpretation.

The *time behavior* subcharacteristic describes the capability of a test specification to provide appropriate response and processing times, as well as throughput rates, when it is instantiated. In testing, this includes especially the necessity to use proper testing and test selection techniques. For example, using a time consuming structural coverage criterion like path coverage can take infinitely long to execute (due to combinatorial explosion), while it does not have any qualitative meaning for the effectiveness of the test.

The *resource utilization* describes the capability of a test specification to utilize appropriate amounts of resources when it is instantiated. Such resources include, for example, human resources necessary in the instantiation. In that sense, the description of test specifications where human interaction is needed can be regarded as bad for the resource utilization and the test specification developer should try to find ways to automate these interactions.

3.3.5. Maintainability

Maintainability of test specifications is important when test developers are faced with changing or expanding a test specification. It characterizes the capability of a test specification to be modified for error correction, improvement, or adaption to changes in the environment or the requirements. The *analyzability*, *changeability*, and *stability* subcharacteristics from ISO/IEC 9126 are applicable to test specifications as well.

The *analyzability* aspect is concerned with the degree to which a test specification can be diagnosed for deficiencies or general properties. For example, test specifications should be well-structured to allow code reviews. Test architecture, style guides, documentation, and generally well-structured code are elements that have influence in the quality of this property. The choice of the specification language plays a role for the analyzability as well. In order to allow easy machine automation of the analysis, the specification language should have a clear and non-ambiguous syntax and its semantics should not be too complex. For example, strongly typed specification eases the automatic analysis as types do not need to be inferred.

The *changeability* subcharacteristic describes the capability of the test specification to enable necessary modifications to be implemented. For instance, badly structured code or

a test architecture that is not extensible may have negative impact on this quality aspect. One example for bad changeability is the usage of magic values, i.e., concrete values in test specifications that occur more than once. If this value would have to be changed as well, all locations where this value occurs need to be changed. A more sensible way to approach such a problem is the use of language elements like constants (if they exist in the specification language).

Unexpected effects due to a modification have negative impact on the *stability* aspect. The stability subcharacteristic is strongly related to the comprehensibility and the analyzability subcharacteristics. The reason is that instabilities due to modifications generally occur if the modified part of a test specification is not well understood by the person who modified it or if the analysis fails and does not uncover the deficiency produced by the modification.

3.3.6. Portability

Portability, the capability of a software product to be transferred from one environment to another, plays only a limited role in the context of test specifications as they are not yet instantiated and thus are neither executable nor bound to a specific platform that they can be transferred from.

Adaptability, however, is relevant to the degree that test specification should be abstract, i.e., it should not be designed for a specific target environment or target SUT. Adaptability prevents the developer, for example, from hardcoding environment specific data, such as IP addresses, port numbers, or login data into the test specification. Similarly, different SUTs may implement different parts, subsets, or extensions of a system specification. Therefore, the test specification must be configurable to allow adaptability regarding such differences.

3.3.7. Reusability

Reusability is the degree to which the test specification or parts of it are reusable to allow easier development of the test specification itself as well as usage for related purposes.

The *coupling* degree is arguably the most important subcharacteristic in the context of reuse. Coupling can occur between test behavior, between test data, and between test behavior and test data. For example, if there is a function call within a test case, the test case is coupled to this function. To make test specifications reusable, the goal is loose coupling and strong cohesion.

The *flexibility* of a test specification is characterized by the length of a specification subpart and its customizability regarding unpredictable usage. For example, if fixed values appear in a part of a test specification, a parameterization may likely increase its flexibility.

Parts of a specification can only be reused if there is a good understanding of the reusable parts, i.e., the *comprehensibility* of the reusable parts must be appropriate. Good documentation, comments, and style guides are necessary to achieve this goal. Interestingly, a high

degree of reusability using techniques like parameterization can sometimes imply less comprehensibility when parts of a specification are expressed in a flexible way. Generalized parts are often more abstract and used more widely. To this extent, quality characteristics that contradict each other have to be balanced well.

3.4. Towards an Instantiation of the Quality Model for Test Specifications

The presented test specification quality model abstracts from the determination of the quality of a test specification with respect to each characteristic and subcharacteristic. To evaluate and assess the quality of a test specification according to the quality model, metrics are associated to the respective characteristics and subcharacteristics for quantification. Related to metrics are bad smells, i.e., patterns of inappropriate code usage. Bad smells are the patterns used to find locations in the test specification with possible problems while metrics are the means to quantify them, for example, the number of inappropriate patterns. To improve those locations of possible misusage, we can apply refactoring. Metrics, bad smells, and refactorings, however, are only the methodical tools needed to recognize, assess, and improve quality problems. They do not provide a methodology that clarifies which metrics to use for the quality assessment. This methodology for metric selection is provided by the Goal-Question-Metric approach. The measurements, i.e., the calculation of metrics and the detection of bad smells in test code, can be performed on two different levels: static analyses refer to analyses and measurements that are performed by looking at the test specification without any form of execution. Dynamic analyses on the other hand attempt to execute the analysis subject in some way in order to analyze properties that cannot be measured statically.

In the following, we introduce methodical tools needed for the quality assessment and metric selection. In addition, we provide a discussion on the difference between static and dynamic analysis for the measurement of software quality.

3.4.1. Software Metrics

The essence of metrics is described in two well-known citations. The first citation was written by Lord Kelvin in 1891:

> "When you measure what you are speaking about, and express it in numbers, you know something about it; but when you cannot measure it, when you cannot express it in numbers, your knowledge is of a meager and unsatisfactory kind; it may be the beginning of knowledge, but you have scarcely, in your thoughts, advanced to the stage of science." [91]

The second citation manages to extract the essence out of Lord Kelvins statement and was phrased by Tom DeMarco in 1982:

> "You cannot control what you cannot measure." [39]

A metric can be regarded as a measurement on a measurement scale—a mapping of a software property into a numerical value or symbol of the measurement scale. A well-known classification for metrics in software development is given by Fenton and Pfleeger [61]. They differentiate between the following kinds of metrics:

- Process Metrics.
- Resource Metrics.
- Product Metrics.

Process metrics are quantifications regarding the process. For example, a metric predicting in how many days a certain milestone is reached concerns the process. *Resource metrics* concern the resources that are used within the project. One such common resource is the human developer. If the manager tried to measure the productivity of the developer, they would be measuring a resource metric. *Product metrics* are concerned with assessing the actual software product. As we are concerned with assessing test specifications, *product metrics* are the most relevant kind of metrics. They are subdivided into metrics that describe:

- Internal Attributes.
- External Attributes.

External attributes are features that are externally visible and can only be measured with respect to how a product relates to its environment. For example, the mean time to failure is an external attribute. *Internal attributes* can be measured in terms of the product itself, e.g., by looking at the code, its structure or its attributes. A metric is *descriptive* when its value can be objectively derived from the code or the specification being measured. By additionally using threshold values, this metric becomes also *prescriptive* [59], i.e., a violated metric threshold indicates a possible quality problem, whereas a metric value below the threshold value is considered to be within acceptable limits.

A vast amount of product metrics of different uses have been proposed in the past. Among them, the well-known *Lines of Code* (LOC) metric that counts the number of code lines with or without comments and empty lines depending on its variant. Other well-known metrics are the cyclomatic complexity from McCabe, which measures the degree to which a piece of code branches in its structure [96] or cohesion and coupling metrics which quantify dependencies between different components or measure the degree to which elements of a code unit belong together [32].

When the goal is to measure and assess test specifications, the product metrics that analyze the test specification prior to its actual instantiation are of special interest, i.e., metrics

that help with the quality assessment early in the development process when an SUT implementation may not be available. Early detected problems can be regarded as predictors or indicators for the problems that can be anticipated when the test specification is instantiated. This can be achieved with metrics that are objective and reproducible. The metrics must have a valid scale and the data must be obtained empirically, i.e., from observations rather than opinion, for example. We assume that problems which are detected and fixed early in the development process (due to measurements) can prevent problems in an instantiated test. For the most part, these metrics are product metrics that measure internal attributes. Such metrics can be generic, test-specific, or language-specific. Generic metrics are applicable to any typical software product no matter which language it is written in. An example for such a metric is the cyclomatic complexity metric, which is applicable to any software product that models a behavioral control-flow. Test-specific metrics refer to properties that are inherent to the nature of test specifications, e.g., metrics that refer to a test verdict. Language-specific metrics are defined with respect to a specific language construct of the language under analysis. For example, metrics measuring properties of templates in TTCN-3 are regarded as TTCN-3-specific.

3.4.2. Software Metrics Selection

The selection of metrics is always a project-specific task, as the concrete measurements depend on the environment, the development process, business goals, or simply how different aspects are emphasized. However, even finding sets of metrics that measure a specific characteristic is not an easy task. In general, the goal is to apply measurements that are both technically and economically easy to achieve. The metrics should be preferably widely accepted. However, especially when weighting different metrics, a common problem is that correlations between metrics are not always evident. Using several non-orthogonal metrics in an evaluation scheme to measure a subcharacteristic may give more weight to certain aspects of the subcharacteristic than using others, and thus subtly influence the objectiveness of the measurement.

For that purpose, different methods have been introduced to evaluate the orthogonality of metrics or metric sets. There are numerous publications that try to tackle the orthogonality problem of software metrics, i.e., they try to identify the measures in a set of metrics that do not deliver any additional meaningful information. One early work of Henry et al. [75] demonstrated the high-degree relationship between the *cyclomatic complexity* [96] and Halstead's complexity measures [71] by means of Pearson's correlation coefficients. A good overview on further related work is provided by Fenton et al. [61]—they list approaches to investigate the correlation of metrics using Spearmans rank correlation coefficient [133] and Kendall's tau rank correlation coefficient [92]. To express the nature of the associations, *regression analysis* (e.g., [30]) has been suggested. Furthermore, *principal component analysis* [118] has been used to reduce the number of necessary metrics by removing principal components that account for little of the variability. A more recent approach tries to tackle

this problem using machine learning [152]. In that method, tuples of a metric set are first used to classify each measured entity of the software under investigation as either “good” or “bad”. This classification is determined by threshold values for each metric in the set. The approach then attempts to approximate the same classification with a smaller set of metrics to reduce the number of necessary metrics in this set and to identify metrics with overlapping information.

Selection of non-orthogonal metrics can still be considered a theoretical problem though that has not been solved for practical purposes. Also, while orthogonality between metrics has been successfully proven, the actual influence or the actual amount of error introduced by intuitively chosen metrics for evaluating certain measureable characteristics remains unknown. Nevertheless, as ISO/IEC 9126 suggests, it is recommended to choose easy to measure, simple, and widely accepted metrics rather than inventing complicated new ones.

A concrete and pragmatic method to determine metrics for the measurement of quality characteristics or subcharacteristics is the *Goal Question Metric* (GQM) approach [8]. GQM is a measurement model that defines three different levels:

1. Goal (the conceptual level): The definition of the goal provides the aim that should be achieved.
2. Question (the operational level): The definition of the question (or questions) provides what should be measured or what questions or properties of the subject the measurement should answer.
3. Metric (the quantitative level): the metric definition provides the solution how to quantify the measurements that answer the questions.

However, GQM continues with three additional steps that provide the actual application, interpretation, and a feedback loop for future improvement:

4. Development of methods and mechanisms for the data collection.
5. Collection, analysis, and interpretation of the data.
6. Post-mortem assessment whether the results conform to the goals. Application of the accumulated experience in future assessments.

Each goal in GQM should be expressed by five different aspects: the object of the measurement (e.g., a test specification module), the aim or purpose of the measurement (e.g., test specification should be future-proof), the quality focus (e.g., reusability), the viewpoint (e.g., perspective of the project manager or the test developer), and the environment (e.g., project or company division). These goals are recorded on so called *abstraction sheets*, which hold additional information, such as quality factors that directly influence the goal, or a hypothesis on quality factors in the assessment subject.

GQM also further specifies detailed items that should be taken into consideration when finding questions for the goal. For example, GQM suggests identifying the meaning of an

answer to a specific question and its possible reactions. Other examples are: weighting the cost/use relationship between the data acquisition or to limit the number of questions to a manageable amount (between three and seven questions).

The GQM approach is pragmatic and easy to understand and apply. When instantiating a quality model according to the GQM approach, the goals are defined on the basis of the nature of the project and its environment. Each project is free to emphasize these quality aspects that are of particular importance. The presented quality model for test specifications provides the necessary model to define the quality factors and the quality focus of the goal. In the following, we present a concrete example of the instantiation of the quality model by applying the GQM method.

3.4.3. Bad Smells

A *bad smell* in software (also called *code smell*) is a term that was first coined by Martin Fowler. Bad smells can be regarded as patterns of possibly inappropriate code usage. Fowler described bad smells as

> "Certain structures in the code that suggest (sometimes they scream for) the possibility of refactoring." [64]

Bad smells are not about errors or bugs in software, but about working pieces of code that are poorly structured. As opposed to a software error where its improvement is a fix of the problem, the improvement of a bad smell in code involves a semantics preserving restructuring or refactoring (see below).

The term bad smell is diffuse in that a location that may be regarded as smelling, may still be the best solution for particular reasons. A good example is code that is optimized for performance. An example for a classic performance optimization is loop unrolling to avoid repeated checks of loop conditions and jumps. While this is an optimization that is usually performed by the compiler today, there are still rare cases where it is still applied manually to code. However, when such code is analyzed, it might be regarded as badly smelling code due to its repetitive nature. Thus, there is still the possibility that certain pieces of code are considered to smell when analyzed even though there may be good reasons why they should not be changed. There may be a cost/benefit analysis involved in the decision whether a smell is to be removed or whether a detected smell is even considered to be one in a project-specific setting.

The notion of metrics and bad smells is not completely disjoint. The occurrences of a bad smell can be counted and thus be interpreted as a metric. On the other hand, if we measure a metric and determine specific threshold values for this metric, a violation of such boundary values may give us a hint regarding a bad smell. Bad smells are usually detected statically, i.e., they are based on the analysis of data structures that are derived from the grammatical structure of the code. However, smell detection and guideline detection (which are relatively

similar, however, guidelines do not try to find code locations that scream for restructuring, but rather check the compliance to certain project standards) can both exhibit cases where the problem is not decidable by static analysis.

3.4.4. Refactoring

Refactoring is the method to improve code or documents that are considered to smell. Refactoring involves the restructuring of this problematic subject without changing the observable behavior. Martin Fowler defines refactoring as follows:

> "A change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior." [64]

The foundation of a refactoring is its systematic description of how a refactoring should be applied. This is achieved by a very strict structure, in which a refactoring is described as well as a mechanical step-by-step instruction for the actual restructuring procedure. The structure of a refactoring description is defined by:

- A name (for building a common vocabulary).
- A summary (for quickly finding refactorings for a specific purpose).
- A motivation (a more detailed description when the refactoring should be applied).
- Mechanics (a systematic procedure how to apply the refactoring).
- Examples.

The mechanics have a particular importance as their intention is to minimize any accidental changes in the semantics. Nowadays, modern *Integrated Development Environments* (IDEs) implement the mechanical steps in order to make the application of refactorings easier, straightforward, and less error-prone. The *Java Development Tools* (JDT) of the Eclipse SDK [46], for example, is well-known for its refactoring capabilities.

3.4.5. Static Analysis

Metrics and bad smells can be determined and analyzed by two different techniques that are as applicable to test specifications as they are general-purpose code: static analysis and dynamic analysis. Static analysis never involves the execution or simulation of a program. Rather, it makes use of code by analyzing the artifacts in their structured form. In the case of programming or textual testing specifications, this structured form is the grammatically structured tree of the software—the abstract syntax tree or any other representation of its abstract syntax.

There are generally two different layers where static analysis can take place: at the level of the specification at hand or using compiled lower-level intermediate code representations, such as byte codes. Lower-level representations are semantically easier to analyze, but they

are more verbose in their form and harder to understand. Also, it is not always easy to draw conclusions from already translated low-level code back to the high-level code. If a problem is found in the low-level code representation, the problem has to be translated back to the terms of the high-level language to enable the identification of the problem.

The most common uses for static analysis within the analytical quality assurance are guideline checking, calculation of metrics, and smell detection. Typical examples for such guidelines are: every function should have an explaining comment, definitions have to take place in a specific order, and similar. Examples for statically measurable metrics are: lines of code, number of statements, counting of syntactical elements (such as functions), or structural measures like the intra-procedural cyclomatic complexity.

Analyzing the abstract syntax of a test specification, or code in general, is influenced by our current theoretical knowledge about computational limits, i.e., certain classes of analyses are reducible to the halting problem [124, 140] and, therefore, cannot be detected using static analysis. In these cases, we are limited to approximations and dynamic analysis approaches.

3.4.6. Dynamic Analysis

Dynamic analysis requires that the software product is executed in some way. Typical use-cases for dynamic analysis are the detection of memory leaks, resource leaks, pointer problems, performance measurements, or even coverage analyses. In general, dynamic analysis opens the possibility to find problems that could be hard or impossible to find by using static analysis.

Technically, there are mainly three distinct ways to dynamically analyze a system:

- Observation of the system at its interfaces or logging facilities.
- Instrumentation of the system.
- Usage of a special-case virtual machine, interpreter, compiler, or processor simulator.

In the first case, the analyzer monitors interfaces or logs facilities for necessary analyses. The analyzer is placed as a man-in-the-middle, i.e., it acts as the software product by providing its interfaces to the environment, intercepting the communication, and forwarding the data to the real software product. The observations are then assessed either on-the-fly (passive testing [10, 31]) or after the execution has finished.

Instrumentation means that the source code of the software product is altered to record additional information about the state of the system in specific parts of the code. What information is recorded and where it is stored depends on the concrete purpose of the instrumentation. Due to the overwhelming amount of work instrumentation would involve when done manually, instrumentation is nearly always performed automatically by a tool. A recent trend is to use the aspect-oriented paradigm to separate instrumentation code from the product code into aspects (for example, [74]).

The third possibility is to move the analysis into the compiler or interpreter that actually runs the code or by in fact simulating the processor that runs the code. Such approaches are useful, for example, to detect memory errors due to misused *malloc* or *free* calls (in C software) or race conditions in multithreaded programs.

There are several different ways to drive dynamic analysis. Some kinds of dynamic analysis are supposed to run in the background and check security constraints or properties while an actual user is interacting with the system. The same can be done in a more target oriented fashion: the developers manually drive the system into states where they expect problems and use the dynamic analysis to support their work. For other purposes, the dynamic analysis often requires that the software product under analysis executes as much different code as possible. A common way to drive the analysis is the execution of test suites against the analyzed software product, as test suites are often developed with some coverage criterion in mind. In these cases, the quality of the analysis result then depends on the quality of the test suite and how well the analysis is tailored for the coverage criterion of the test suite. Model checking can also be regarded as a special kind of dynamic analysis. Here, the behavior of the software under analysis is analyzed for properties in as many execution paths as possible, while trying to minimize the actual number of analyzed paths that are probed.

Dynamic analysis (except for model checking) suffers from the problem that only executes the code that it is targeted to execute. In particular, when parallel and distributed software is analyzed dynamically, there is no guarantee that the same behavior is executed as in previous runs, even if the stimuli that are sent to the analyzed system are the same. Even if the timing is different, we are in practice already dealing with different global states. This exposes the problem that on the one hand it might not be easy to reproduce occurring problems for the dynamic analysis and on the other hand, it may be hard to repeat a dynamic analysis.

3.5. An Instantiation for TTCN-3 Projects

The instantiation of the test specification quality model requires a set of metrics for each subcharacteristic that adequately capture the different quality aspects of a project in numbers. There are various ways to obtain these numbers: static analysis, dynamic analysis on the specification level, but also results from manual reviews of specification documents. The latter may include the comparison of different kinds of test specification documents to assess the degree of consistency between them or the coverage of specifications.

Following the GQM approach, we define the goal to be the improvement of the respective quality characteristics. In the following, we provide an exemplary instantiation of the quality model for test specifications for a test project using the TTCN-3 language for a selection of characteristics and subcharacteristics. Figure 3.4 depicts what quality characteristics and subcharacteristics have been selected for the exemplary instantiation. It is structured as follows: each section instantiates a specific characteristic and subcharacteristic denoted by the

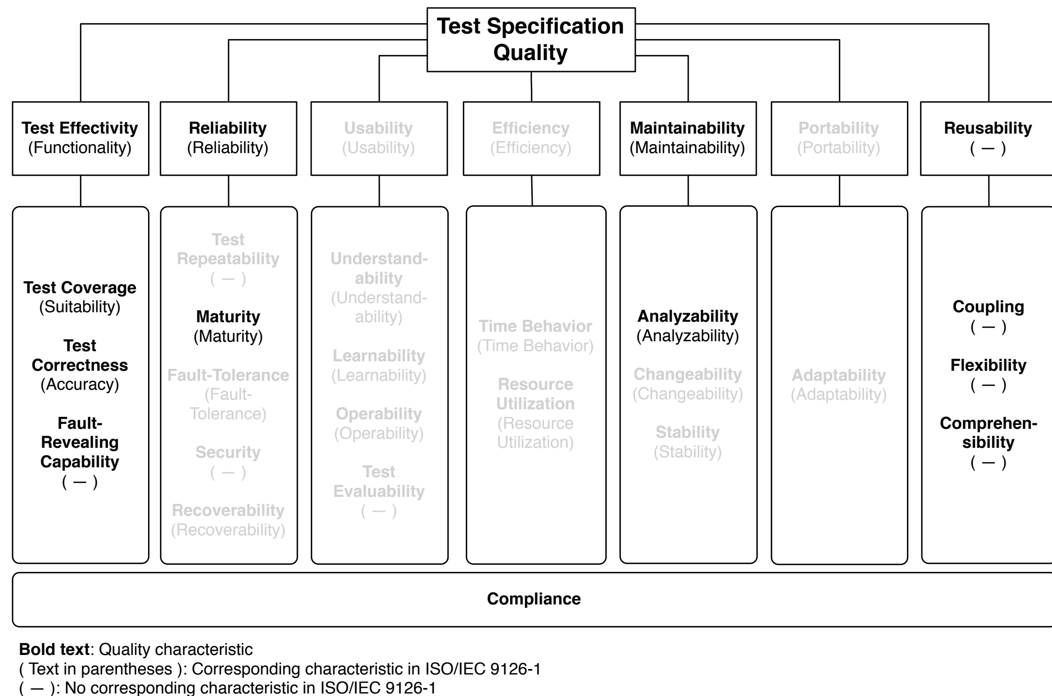


Figure 3.4.: Instantiated Parts of the Quality Model for Test Specifications

notation *characteristic / subcharacteristic*. For each subcharacteristic, we provide a number of questions. Finally, we provide metric definitions that deliver answers to the questions on a concrete scale. These can be either generic, test-specific, or specific to the TTCN-3 language. The exemplary instantiation will show that the actual determination of the metrics is not hard once the goal has been specified and the right questions have been asked. As mentioned before, the instantiation is project-specific and therefore, this concrete instantiation is not a generalization that applies to all projects even though the questions and metrics have been chosen to be rather general. Based on the results of the assessments, refactorings can be applied to improve the quality characteristic.

3.5.1. Characteristic: Test Effectivity / Test Coverage

Answers to the following questions deliver an assessment for the test coverage subcharacteristic of the test effectivity characteristic:

- To what degree are the test purposes covered by the test specification?
- How well does the test specification test the SUT?

The following metrics provide answers to these questions:

Definition 3.1 (Metric: Test Purpose Coverage)

$$\textit{test purpose coverage} := \frac{\textit{number of test purposes covered by TTCN-3 test cases}}{\textit{overall number of test purposes}}$$

The number of test purposes covered by test cases specified in a TTCN-3 test suite is compared to the number of test purposes contained in a corresponding test purpose specification. The metric is test specific.

Definition 3.2 (Metric: System Model Coverage)

$$\textit{system model coverage} := \frac{\textit{test coverage of system model}}{\textit{possible coverage of system model}}$$

Several different coverage criteria like path coverage, branch coverage, etc. are applicable. This metric determines the test coverage with respect to a model of the SUT. The metric is test specific.

3.5.2. Characteristic: Test Effectivity / Test Correctness

Answers to the following questions deliver an assessment for the test correctness subcharacteristic of the test effectivity characteristic:

- Does the test case deliver consistent test verdicts?
- Does the test case terminate?

The following metrics provide answers to these questions:

Definition 3.3 (Metric: Test Verdict Completeness)

$$\sigma = \textit{number of paths in TTCN-3 test cases setting no test verdict}$$

$$\textit{test verdict completeness} := \begin{cases} 1 & \textit{if } \sigma > 1 \\ 0 & \textit{otherwise} \end{cases}$$

It is assessed whether there are any paths that do not set a test verdict. The metric is test specific.

Definition 3.4 (Metric: Early Test Verdict)

$$\sigma = \textit{number of paths in TTCN-3 test cases setting a test verdict before any communicating behavior}$$

$$\textit{early test verdict} := \begin{cases} 1 & \textit{if } \sigma > 1 \\ 0 & \textit{otherwise} \end{cases}$$

It is assessed whether test verdicts are set prior to any communication. This may indicate a possible problem or design weakness, especially if the test verdicts are not *pass*. The metric is test specific.

Definition 3.5 (Metric: Test Termination)

$$\textit{test termination} := \frac{\textit{number of paths in TTCN-3 test cases terminating correctly}}{\textit{overall number of paths in TTCN-3 test cases}}$$

It is assessed whether all paths of the test cases terminate correctly. The metric is test specific.

3.5.3. Characteristic: Test Effectivity / Fault-Revealing Capability

Answers to the following questions deliver an assessment for the fault-revealing capability subcharacteristic of the test effectivity characteristic:

- Is the test data thoroughly evaluated?
- Are all effects of the system covered?

The following metrics provide answers to these questions:

Definition 3.6 (Metric: Template Transmissibility)

$$\sigma = \textit{number of wildcard-only-covered elements in type definitions used by receiving templates}$$

$$\rho = \textit{overall number of elements in type definitions used by receiving templates}$$

$$\textit{template transmissibility} := 1 - \frac{\sigma}{\rho}$$

SUT responses might never be properly evaluated when the corresponding receiving templates are too transmissible due to wildcards. This metric measures to what degree the elements of received data are at least covered once by a non-wildcard expected value. As the template concept is specific to the TTCN-3 language, the metric is TTCN-3-specific.

Definition 3.7 (Metric: Effect Coverage)

$$\textit{effect coverage} := \frac{\textit{number of effects tested by a TTCN-3 test suite}}{\textit{overall number of effects possible in the system specification}}$$

This metric uses cause-effect analysis to determine the degree to which each effect is at least tested once. The metric is test specific.

3.5.4. Characteristic: Test Reliability / Maturity

An answer to the following question delivers an assessment for the maturity subcharacteristic of the test reliability characteristic:

- Are there multiple timeout branches in an alt-statement where one timeout is concrete and the other is an any timeout?

The following metric provides an answer to this question:

Definition 3.8 (Metric: Timeout Inconsistency)

$$\begin{aligned}
\sigma = \; & \textit{there is a path where a timer is started and} \\
& \textit{where subsequently a directly referenced timeout} \\
& \textit{branch takes place} \\
\rho = \; & \textit{there is a corresponding path where} \\
& \textit{in the same branch an any timeout branch exists} \\
\textit{timeout inconsistency} := \; & \sigma \wedge \rho
\end{aligned}$$

In particular, when defaults are activated that globally catch all timeouts using the *any timer* reference, it is not clear to developers that this implicit timeout exists due to the default exists and they implement their own timeouts. The implementation of an explicit timeout referencing the timer may be developed on purpose, for example, because the verdict handling or follow up behavior must be different, or it is a redundancy due to the developer not realizing that such a default is activated. As the timer concept is specific to the TTCN-3 language, the metric is TTCN-3-specific.

3.5.5. Characteristic: Maintainability / Analyzability

An answer to the following question delivers an assessment for the analyzability subcharacteristic of the maintainability characteristic:

- How complex is the test case?

The following metric provides an answer to the questions:

Definition 3.9 (Metric: Complexity Violation)

$$\sigma = \textit{number of behavioral entities violating upper bound of complexity}$$

$$\rho = \textit{overall number of behavioral entities}$$

$$\textit{complexity violation} := 1 - \frac{\sigma}{\rho}$$

This metric measures the number of TTCN-3 testcases, functions, and altsteps which violate a defined boundary value of a complexity measure in comparison to the overall number of testcases, functions, and altsteps. Several complexity measures may be used, e.g., McCabe's cyclomatic number [96], the nesting level, the call-depth, or the number of statements. The metric is generic.

3.5.6. Characteristic: Maintainability / Changeability

Answers to the following questions deliver an assessment for the changeability subcharacteristic of the maintainability characteristic:

- How often is the same code repeated?
- How strongly is a given entity coupled to other test code pieces in the test specification?
- Are there unused or rarely referenced definitions?
- Are there similar test data definitions?

The following metrics provide answers to these questions:

Definition 3.10 (Metric: Code Duplication)

$$\textit{code duplication} := 1 - \frac{\textit{entities containing duplicated code}}{\textit{overall number of entities}}$$

Since changes to duplicated code require changing all locations of duplication, this metric determines the portion of duplicated code in terms of, e.g., LOC or statements. The metric is generic.

Definition 3.11 (Metric: Maximum Number of References Violation)

$$\sigma = \textit{number of entities which are referenced more times than an upper bound allows}$$
$$\rho = \textit{overall number of entities}$$

$$\textit{maximum number of references violation} := 1 - \frac{\sigma}{\rho}$$

This metric determines how often an entity is referenced and penalizes the violation of an upper boundary value. When applying changes to entities which are referenced very often, a developer needs to check each reference whether a change may have unwanted side effects or whether it requires follow-up changes. The metric is generic.

Definition 3.12 (Metric: Removable Definition)

$$\textit{removable definition} := \begin{cases} 0 & \textit{number of references to the definition} > 1 \\ 1 & \textit{definition is not referenced or referenced only once} \end{cases}$$

This metric determines whether a definition is never referenced or referenced only once. A definition that is never referenced may be removed. A definition referenced only once may be inlined, for example. The metric is generic.

Definition 3.13 (Metric: Similar Template)

$$\textit{similar template} := \begin{cases} 1 & \textit{percentage of differing fields} < 25\% \\ 0 & \textit{otherwise} \end{cases}$$

This metric determines whether two template definitions are similar. As the template concept is specific to the TTCN-3 language, the metric is TTCN-3-specific.

3.5.7. Characteristic: Maintainability / Stability

Answers to the following questions deliver an assessment for the stability subcharacteristic of the maintainability characteristic:

- Are there any global variables?
- Are there possible side effects?

The following metrics provide answers to this question:

Definition 3.14 (Metric: Global Variable and Timer Usage)

$$\sigma = \textit{number of component variables and timers referenced by more than one behavior}$$

$$\rho = \textit{overall number of component variables and timers}$$

$$\textit{global variable and timer usage} := 1 - \frac{\sigma}{\rho}$$

Global variables promote side effects. In TTCN-3, component variables and timers are global to all behavior running on the same component. This metric measures the number of all component variables and timers referenced by more than one function, testcase, or altstep and relates them to the overall number of component variables and timers. As the timer concept is specific to the TTCN-3 language, the metric is TTCN-3-specific.

Definition 3.15 (Metric: Parameter Amount with Possible Side Effects)

$$\textit{parameter amount with possible side effects} := 1 - \frac{\textit{number of out and inout parameters}}{\textit{overall number of parameters}}$$

Any modification of parameters which are passed into a testcase, a function, or an altstep as *out* or *inout* parameter may lead to side effects. Hence, this metric measures the possibility of side effects regarding the parameters by relating the number of out and inout parameters to the overall number of parameters. The metric is generic.

3.5.8. Characteristic: Reusability / Coupling

Answers to the following questions deliver an assessment for the coupling subcharacteristic of the reusability characteristic:

- To what degree is a module coupled to other modules?
- To what degree can existing behavior be generalized?

The following metrics provide answers to these questions:

Definition 3.16 (Metric: Coupling to Other Modules)

$$\textit{coupling to other modules} := \frac{\textit{number of modules importing from other modules}}{\textit{overall number of modules}}$$

The reusability of the modules of a test suite depends on how tightly each module is coupled to other modules. Hence, this metric counts the number of modules coupled to other modules and relates this to the overall number of modules. In TTCN-3, coupling between modules is introduced by the *import* construct. As many languages provide some sort of import mechanism, the metric can be considered to be generic.

Definition 3.17 (Metric: Coupling to Overspecialized Components)

$$\sigma = \textit{number of behavioral entities unnecessarily running on a specialized component}$$

$$\rho = \textit{overall number of behavioral entities running on components}$$

$$\textit{coupling to overspecialized components} := \frac{\sigma}{\rho}$$

Reusability is reduced if a function, a testcase, or an altstep running on a component would run on a parent component as well, but is bound to a more specialized component. Hence, this metric relates the number of such cases to the overall number of functions, testcases, and altsteps which are coupled to components in general. As the component specialization concept is specific to the TTCN-3 language, the metric is TTCN-3-specific.

3.5.9. Characteristic: Reusability / Flexibility

An answer to the following question delivers an assessment for the flexibility subcharacteristic of the reusability characteristic:

- Does the current implementation of a behavioral entity endanger its reuse?

The following metrics provide an answer to this question:

Definition 3.18 (Metric: Entity Shortness)

$$\textit{entity shortness} := \frac{\textit{number of behavioral entities violating a given size limit}}{\textit{overall number of behavioral entities}}$$

The shorter an entity is, the higher the probability is that it is flexible enough to be reused in a different context. Hence, this metric measures the number of testcases, functions, and altsteps whose LOC or number of statements violate a defined boundary value. The metric is generic.

Definition 3.19 (Metric: Parameterization Degree)

$$\textit{parameterization degree} := \frac{\textit{number of formal parameters}}{\textit{number of formal parameters} + \textit{number of hardcoded values}}$$

Reuse is hindered by hard-coded values and promoted by parameterization. Hence, this metric values parameterization and penalizes hard-coded values. The metric is generic.

3.5.10. Characteristic: Reusability / Comprehensibility

Answers to the following questions deliver an assessment for the comprehensibility sub-characteristic of the reusability characteristic:

- How well are entities documented?
- How well are the entities structured?

The following metrics provide answers to these questions:

Definition 3.20 (Metric: Annotation Degree)

$$\textit{annotation degree} := \frac{\textit{number of commented entities}}{\textit{overall number of entities}}$$

The number of entities, e.g., testcases, functions, or altsteps, whose interface is properly documented in comparison to the overall number of considered entities. The metric is generic.

Definition 3.21 (Metric: Groupedness)

$$groupedness := 1 - \frac{number\ of\ ungrouped\ elements}{overall\ number\ of\ elements}$$

In TTCN-3, grouping is a means to structure the elements of a module. Hence, this metric calculates the degree of structuredness by penalizing unstructured elements. As the group concept is specific to the TTCN-3 language, the metric is TTCN-3-specific. However, other languages may provide similar concepts.

3.5.11. Characteristic: Compliance

An answer to the following question delivers an assessment for the compliance characteristic:

- How many non-timeout paths set a verdict before stopping the timer?

The following metric provides an answer to this question:

Definition 3.22 (Metric: Verdict/Timer Inconsistency Degree)

$$\begin{aligned} \sigma = &\ number\ of\ non\text{-}timeout\ paths\ where\ a\ verdict \\ &\ is\ set\ before\ the\ timer\ is\ stopped \\ \rho = &\ overall\ number\ of\ non\text{-}timeout\ paths \\ verdict/timer\ inconsistency\ degree := &\ \frac{\sigma}{\rho} \end{aligned}$$

If there are paths where a timer is started and in which the timer does not timeout, we expect that this timer is stopped prior to the determination of any test verdict.

3.6. Related Work

So far, there is no holistic approach towards the quality assurance of test specifications. For the most part, there are only techniques and mechanisms that address step 4 of the GQM approach, without a predefined goal. Vega, Din, and Schieferdecker have worked on guideline checking of TTCN-3 test suites in order to improve the maintainability of test suites [43]. They classify guidelines for TTCN-3 on three different levels: the architectural level, the language level, and the physical level. The architectural level refers to information related to the SUT, e.g., interfaces, use-cases, or roles. A guideline for this level would be, for

example, to group together definitions belonging to an SUT interface, specific roles, or use-cases. The language level refers to the definition of test constructs of TTCN-3, such as types, components, test cases, and similar. Guidelines for this level are formatting rules, naming conventions, and structural rules. The physical level deals with file system information such as files and folders. Guidelines for this level may, for example, provide information on how to structure the test suite in folders within the file system. Information from the architectural level can be propagated to the language level and from there to the physical level. Information from the language level can be propagated to the physical level. Some guidelines can be interpreted as metrics with predefined threshold values. Even the guideline checking of naming conventions can be interpreted as metrics—they measure the amount of violations where specific identifiers do not fulfill a convention and report the affected locations within the test specification. These three specified levels are also valid from the point of view of the quality model presented in this chapter. These three levels are arranged on a different axis than the quality characteristics specified in the quality model for test specifications, i.e., any quality characteristic or subcharacteristic of the quality model for test specifications may be assessed on these three different levels. Therefore, the classification into architectural level, language level, and physical level is relevant also for the instantiation of the quality model.

Besides guideline checking, the same authors have worked on quality measures for test data [147, 149]. This first work on test data quality discusses test data variance as a measure for test data quality, where test data variance is defined as the test data distribution over the system interface data domain. The underlying idea is that a higher degree of test data variance implies also a higher test quality. In other work on test data, they evaluate the quality of test data stimuli. For that, a data clustering method to partition types and subsequently measure the coverage related to the data clusters is presented. They have applied this technique to TTCN-3 types and used constraint programming over the control-flow graph to deal with variable templates. In this thesis, we will not deal with test data and how it related to the quality of a test specification, but concentrate on behavioral aspects of test quality. However, it is true that test data is a very important factor in the assessment of test quality and, therefore, these results are very relevant for the quality assessment of test specifications in general. Vega, Din, Schiefedecker et al. have also suggested concrete TTCN-3-specific quality metrics [148]. However, they do not provide an underlying quality model. With the TRex tool [154], there exists a refactoring catalog and prototype for improving issues in TTCN-3 test suites. Sneed, Baumgartner, and Seidl published a book on system testing [132] which briefly mentions the topic of quality assurance for test cases. Several metrics for test case quality are proposed in this work such as test case complexity based on the density of test data, test data complexity, test data volume, or test case density. To measure test quality, they propose metrics for test case impact (i.e., which entities are affected by the test), test case modularity, test case completeness, and test case reusability. These kinds of test metrics are defined in more general terms and in fact rather define the questions that need to be asked within the GQM approach than providing answers to concrete questions. Interpreted as questions though, the metrics are relevant and can be related to characteristics and subcharacteristics of our quality model.

Mutation testing approaches (such as the more recent work from Schuler and Zeller [130]) can be considered as an analytical approach for the assessment of the fault-revealing capability of test suites and can, therefore, be related to our quality model. By changing the SUT with non-equivalent mutants, the idea is to uncover these mutants with the test suite. If all mutants are uncovered by the test suite, it can be considered of high quality. If a mutant slips through and is not detected, the test developers gain knowledge where they may extend their test suite and where test cases are missing. So far, mutation testing has only been applied in very small scale case studies. Schuler and Zeller are among the first who have applied mutation testing to projects of realistic size and complexity.

Tool support for static analysis is well developed for many commonly used languages. In particular, there is a strong support for strongly typed languages, like Java or C#, regarding static analysis tools. Popular tools for Java are FindBugs [7], PMD [120], or Checkstyle [1]. All these tools perform static analysis and have sets of predefined rules and guidelines. Typical examples are the checking of naming conventions and formatting guidelines, but also data-flow analyses to identify locations where an uninitialized variable may be referenced. Most of these tools include some kind of integration into popular IDEs, such as Eclipse [46], that allow continuous on-the-fly observation of the code for possible problems by running transparently in the background and alerting when problems are found. It is noteworthy that all these tools are practically usable in everyday work and have matured well. However, besides our own TRex tool, we are currently not aware of any other tooling that supports static test quality analysis for test-specific properties. Related work regarding dynamic analysis and dynamic analysis of test specifications is discussed in Section 4.6.

4. Model-Based Analysis of Test Specifications

Dynamic analysis requires that the software product is executed in some way. Use-cases for dynamic analysis are the detection of memory leaks, resource leaks, pointer problems, performance measurements, or even coverage analysis. An observation is that the uses for dynamic analysis are often of a more technical nature although there are also attempts to use dynamic analysis to generate models for documentation purposes [62]. In general, dynamic analysis opens the possibility to find problems that could be hard or impossible to find using static analysis.

In the following chapter, we describe dynamic analyses for test specifications. These dynamic analyses are all based on models that are constructed from test cases. Such analyses can be considered as a special form of testing of test specifications. In practice, we can only check certain aspects of the test behavior. Even though test specifications may contain errors like any other kind of software specification, it is common sense that it is not feasible to write test specifications for test specifications. After all, we would be stuck in a vicious circle as test specifications for test specifications could be again error-prone. Instead, we need to find ways to check the test specification in a more generic way. For this purpose, we define generic properties that should always or never be true for test behavior models.

Figure 4.1 depicts the overall analysis method presented in this chapter. First, a test case specification is simulated. The events produced during the simulation are the input for a re-

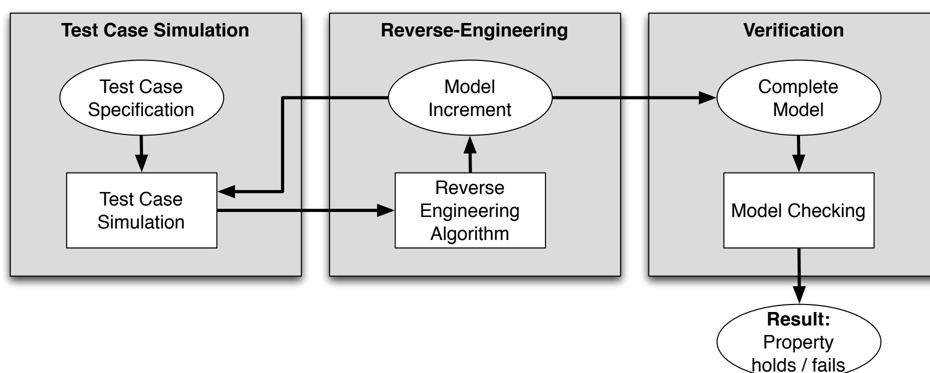


Figure 4.1.: Dynamic Analysis Methodology

verse engineering algorithm that produces model increments for each set of events coming from the test case simulator. If the test case simulation is finished, we consider the current model increment to be the completely reconstructed model. This model is then model checked for certain properties, for example, test-specific properties that analyze the consistency of test verdicts. The model checking then concludes either that the property holds or that it fails. In the latter case, the model checker delivers a failing trace that describes the behavior where the property does not hold. This information can then be mapped back to the test case specification to examine the possible problem in the test specification language. The presented method is general, i.e., it can be applied to TTCN-3, but it does not make use of any specific TTCN-3 features and we expect the method to work for other languages as well.

In the first section of this chapter, we discuss reverse engineering (Section 4.1). Subsequently, we present a formal model for describing test specifications (Section 4.2). Our model reconstruction[4] method is described in Section 4.3. The reconstructed models are then analyzed. A catalog of generic properties that can be applied to test specifications can be found in Section 4.4. Following our test case analysis, we shift our focus to test suite properties and discuss one specific test suite anomaly that we call *response consistency*. Finally, we conclude with a discussion of related work (Section 4.6).

4.1. Reverse Engineering

Reverse engineering in general is the process of analyzing an object in order to find out how it works. Reverse engineering, however, is a very diverse term and while the generic meaning of it almost always is true to some degree, people may understand vastly different things confronted with this term, depending on what their expertise and domain are. One group of people might associate the term to reversing a binary program from machine code into more readable (albeit still very low level) assembler code—possibly with the goal to patch a small part of it. Other persons might associate the term with building UML class diagrams from their Java code. Chikosfky tried to solve the confusion by defining key terms in 1990 [33]. He defines reverse engineering as follows:

> "Reverse engineering is the process of analyzing a subject system to
>
> - Identify the system's components and their interrelationships and
> - Create representations of the system in another form or at a higher level of abstraction."

Reverse engineering never involves the alteration of a system, it is merely an examination process that usually involves the extraction of designs and building less implementation-dependent abstractions of it. A common use for reverse engineering is, for example, the

[4] We use the terms *model reconstruction* and *model reverse engineering* synonymously.

generation of documentation in order to make a system (or legacy system) easier to understand for people who did not develop it. It can be performed on any level of abstraction in any phase of the software development process.

4.1.1. Documentation and Design Recovery

Two common areas of reverse engineering are documentation and design recovery. The goal of documentation recovery is the automatic recreation of the documentation for a system that should have existed in the first place. The documentation represents a different view towards the system which is oriented towards a different audience than someone who works with the actual code. For example, the generation of UML diagrams from code is such a documentation recovery application of reverse engineering. Another different use would be the generation of cross-reference documentation [115, 145].

In design recovery, in addition to the actual observations or code from the object, several sources of knowledge (for example, documentation, personal knowledge about the object) may be combined to deduce and reason about higher level abstractions. It aims at recovering as much information about a system beyond the information that is directly provided from the object.

4.1.2. Reengineering

An important term that needs to be clearly distinguished from reverse engineering is *reengineering*. The terms reverse engineering and reengineering are sometimes confused with each other or even used interchangeably. Chikovsky defines reengineering as follows:

> "Reengineering, also known as both renovation and reclamation, is the examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form." [33]

Reengineering may include reverse engineering within the process to reconstitute a system in a new form, for example, to extract information from a legacy system. Using the extracted information and parts of the old system, reengineering denotes the restructuring of the old system with possibly some forward engineering. Here, reverse engineering helps in the creation of a system that is modern or that may be changed to realize new requirements that are not met by the original system.

4.1.3. Recovering and Reengineering Artifacts of Development Phases

Figure 4.2 illustrates the mapping of forward and reverse engineering to the development phases (in this case, the phases from the V-Modell 97 [60] with an additional binary code layer). In the direction of the left hand side to the right hand side, the conventional kind of development takes place in the form of forward engineering. The direction from right hand

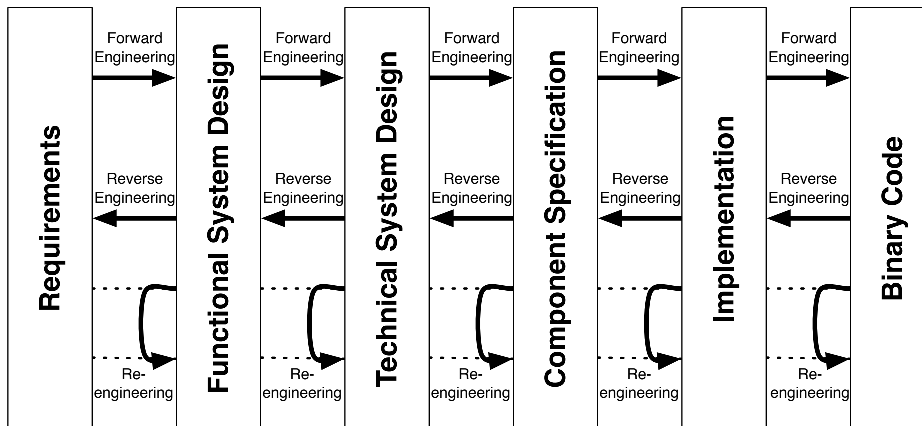


Figure 4.2.: Forward and Reverse Engineering in the V-Modell 97 Phases

side to left hand side is the reverse engineering direction. For example, if people intend to get the implementation from binary code, they reverse engineer the binary code using a disassembler. The artifacts produced in other phases can be reversed comparably. The result is always the artifact of the phase on the left side of artifacts' phase that we reverse. If some kind of change, alteration, or any other kind of forward engineering takes place with the help of a reverse engineered artifact, we calling it reengineering.

The work on reverse engineering is as manifold as the different interpretations of the term. We have disassemblers for binary code like IDA Pro [127], decompilers for Java bytecode like the Java Decompiler [2], UML tools like Magicdraw [107] which create class and sequence diagrams from code in different languages, the Javadoc tool [115], or work where object state machines are reverse engineered from branch-covering unit test executions [153].

4.2. A Formal Model for Describing Test Case Specifications

In Section 2.2, we introduced formal notations for describing transition systems. However, it is not yet clear how we model test cases and test suites in terms of these definitions. As a formal model is required for the analyses in this chapter, this section provides the missing link between the formal model notations of Section 2.2 and the model that we will use to actually represent the test cases and test suites that we intend to analyze. In this section, we first provide a formal definition for test case and test suite models. Afterwards, we provide an example how these kinds of models look like, also in relation to TTCN-3 test cases, and discuss the adequacy of these models for the analysis of test behavior in general and for representing TTCN-3 behavior in particular.

4.2.1. Test Case Model

The kinds of test cases that we intend to represent have the following properties:

- The test cases may have multiple components with behavior that may run in parallel.
- The test cases communicate with each other and the SUT by means of messages sent via channels.
- The communication is non-blocking, i.e., a message sent to another system is always received in an input queue and the sending component can immediately carry on with its behavior.

With these preconditions in mind, we can define a test case as follows:

Definition 4.1 (Test Case) *A non-blocking test case T with queues is a tuple (C,P,ρ) where*

- *C is the set of test components modeled as LTS or EMIOTS that each describe the behavior of one test component in a test case.*
- *P is a set of ports.*
- *ρ is a relation that assigns ports to a test component C, i.e., $\rho \subseteq P \times C$ where $(p,c) \in \rho$ implies that there is no c' with $(p,c') \in \rho$.*

The composite behavior of the test case is then defined as $(c_1 \; \|_Q \; q_1) \; \|_M \; (c_2 \; \|_Q \; q_2) \; \|_M \; \dots \; \|_M \; (c_n \; \|_Q \; q_n)$ with $c_i \in C$, $0 \leq i \leq n$. The queues $q_1,\dots,q_n$ are defined by the ports assigned to each component, i.e., for each $(p_j,c_i) \in \rho$, there must be a queue q_j that accepts messages on port p_j. The overall queue behavior q_i of a component c_i is then defined by the parallel composition of all q_j.

Example: we have a test case with two test component EMIOTS c_1 and c_2. We have a total of three ports p_1, p_2, and p_3. Port p_1 and p_2 are assigned to component c_1 and port p_3 is assigned to component c_2. We then have EMIOTS queues q_1, q_2, and q_3 corresponding to each port. The overall behavior of this system is then described by $(c_1 \; \|_Q \; (q_1 \; \|_M \; q_2)) \; \|_M \; (c_2 \; \|_Q \; q_3))$.

Having defined and described what a test case is, we can complete our set of definitions by defining a model for *test suites*.

Definition 4.2 (Test Suite) *A test suite is a set TS of test case models T.*

In some cases, it might be useful to define the test suite as an ordered set rather than an unordered set. For our purposes, an unordered set is sufficient.

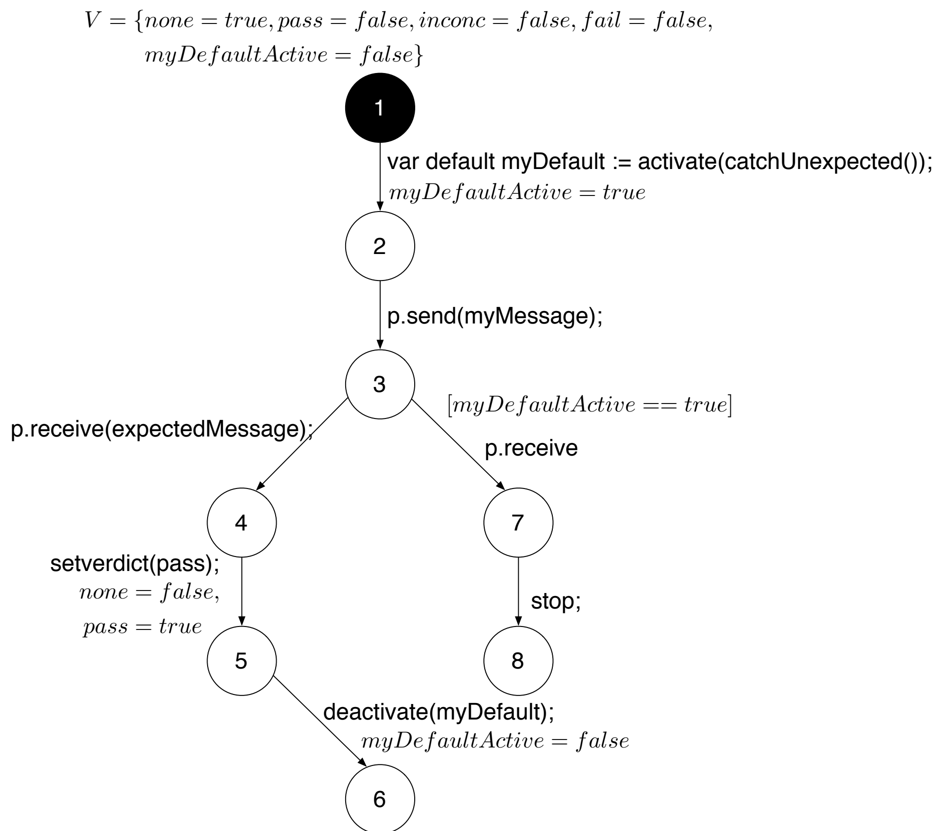


Figure 4.3.: Example: TTCN-3 Test Case to EMIOTS Component Mapping

4.2.2. Test Case Model Example

In the following, we present an example of what a test case model looks like and how it may relate to a piece of TTCN-3 code. For that purpose, we show how to represent the test case depicted in Listing 2.2 of Section 2.1.6. Figure 4.3 illustrates a mapping of the behavior of the test case running on one component to an EMIOTS (also representing the behavior of one test component of the test case model). In essence, the inter-procedural control-flow of the TTCN-3 test case is modeled. In this example test case, we model two kinds of data: the verdicts and the default. In this case, it is sufficient to model the data using boolean variables. In the start state, the *none* verdict is set to true while all other verdicts are initialized to false. The variable describing the default activation *myDefaultActive* is initialized to false. Upon activation, its value is changed to true. Similarly, the verdict variables are changed in the *setverdict(pass)* action between state 4 and 5. Finally, the transition from state 3 to state 7 is guarded by the value of the *myDefaultActive* value.

The example presents an intuitive mapping of TTCN-3 behavior to an EMIOTS. We describe a thorough model reconstruction method in Section 4.3. The overall model describing the complete test case behavior is in fact more complex than the model presented in the figure. The reason is that the illustration omits the parallel composition with the queue model for port *p*. As parallel compositions yield very complex models, we present for the most part only models representing isolated test components of composite test case models for our discussions in this thesis. In subsequent illustrations, we also use the shortcuts for describing message-based communication, for example, we use *p?expectedMessage* instead of *p.receive(expectedMessage)*.

Note that the test case terminates without a verdict if an unexpected message is received—as no verdict is set, the resulting verdict would be *none*. This is clearly not a desirable result for a test case and likely a mistake that the test developer made. Nevertheless, the problem is not that easy to detect: first, we need an interprocedural analysis of the behavior as the altstep is not part of the actual test case. Second, it is statically not decidable whether the default altstep is attached to the alt-statement or not for all behavioral cases. With the method presented in this chapter, we intend to make such cases visible, where a specific property—in this case, a missing test verdict in case of unexpected behavior—is violated.

4.2.3. Test Case Model Adequacy and Abstraction

In the previous two subsections, we have defined a test case model. However, it is unclear at this point to which degree the model is suitable for representing test case behavior in general and TTCN-3 behavior, the language that we later use in our case study, in particular. In general, the operational semantics of program behavior is typically described using a transition system to which syntactical constructs of the language are mapped. One such formalism that is based on transition systems is the *Structural Operational Semantics* (SOS) pioneered by Plotkin [119]. As both the LTS and EMIOTS models, on which the test case model is based, are supersets of a classic transition system (consisting only of configurations and transitions), by only carrying additional information, the models are suited for the representation of program behavior in general. The test case model is specified on a higher level than the classic transition system (by usage of variables and guards) and describes how the message-based communication paradigm and the non-blocking behavior are composed using multiple such models. However, the result is still an LTS or EMIOTS respectively.

The test case model has been designed to cover the main paradigms of the communicating behavior of TTCN-3. The TTCN-3 behavior itself is described using a kind of high-level transition system, a flow graph, in the respective operational semantics document [56]. The TTCN-3 behavior is defined by nodes which carry attributes, such as lists with the active default altsteps, timer states, or value stacks. A set of dependent rules then describe how syntactical elements modify the attributes in these nodes, as well as how and when different paths in the flow graph are taken. Like other operational semantics formalisms, it describes how the complete set of syntactical elements of TTCN-3 has to be interpreted at runtime.

The flow graph of the operational semantics of TTCN-3 is different from the test case model that we use in this thesis to represent test case behavior. The first difference is that the EMIOTS model is rather low-level and its semantics essentially amount to the evaluation of boolean variables in the transition guards. The TTCN-3 flow graph on the other hand is a higher-level model that includes structural features, such as reference nodes, which are effectively mapped to lower level constructs. For example, in case of a node reference, it is replaced by the respective graph segment that it references. The semantics also has its own formalism attached to the nodes and specifies how attributes are manipulated. The way the flow graph semantics is described eases the understandability of the TTCN-3 program execution in comparison to a formal semantics description. Yet, it is a theoretic construct that supports language engineers in writing tools for the language. On the other hand, the EMIOTS test case model does not describe how to interpret TTCN-3 behavior, rather it presents the structure of the behavior—not for a specific execution, but as a representation of the overall behavior.

Dynamic properties, as presented in this chapter, could in fact be expressed over the entity states of the operational semantics in TTCN-3. However, this would bind the expression of the dynamic properties to TTCN-3, whereas the properties that we present later, may also be applied to other testing languages with similar paradigms. The presented test case model is abstract enough to fit other languages as well. To model data, we augment the model with those attributes (in our case variables in the EMIOTS) that we are interested in. When we analyze TTCN-3, the EMIOTS variables in fact refer to attributes of the entity states of the TTCN-3 operational semantics. The amount to which entity state attributes are mapped to the EMIOTS variables determines the level of abstraction involved. In the following sections, we map the control-flow of the test behavior completely to the EMIOTS, whereas the data-flow will be mapped partially. In that sense, we deal only with data abstraction.

The data-flow information is necessary for variable-based verification and behavior restrictions. One the one hand, data-flow is used for checking actual properties that are related to the data values and their flow through the behavior. On the other hand, they restrict behavior. For example, when the value of a variable is checked by a guard, conditional behavior, or even a loop, then this variable check reduces the amount of possible behavior through its dependence on the data values. As a result, disregarding data-flow in behavior implies an abstraction to the behavior, in the sense that more behavior is available as opposed to when data-guarded behavior restrictions apply.

If we introduce abstractions that allow more behavior (e.g., by disregarding certain data-flows and data-based guards), a safety property that does not hold on the reconstructed model does not make any statement about the original behavior (so called false positives). On the other hand, if a safety property holds on a model that has more behavior than the original model, then it a fortiori holds for the behavioral subset as well. The terms logical soundness and logical completeness deliver important aspects for this discussion.

Definition 4.3 (Logical Soundness) *An abstraction α is logically sound with respect to a behavioral model M and a property ϕ if for any execution σ of M violating ϕ, there exists a corresponding execution σ' in $\alpha(M)$ that violates ϕ.*

Definition 4.4 (Logical Completeness) *An abstraction α is logically complete with respect to a behavioral model M and a property ϕ if for any execution σ' of the abstract model $\alpha(M)$ violating ϕ, there exists a corresponding execution σ in M that violates ϕ.*

In this chapter, we concentrate on a model that is adequate for structural analyses, i.e., analyses that are somehow related to the flow of events that are occurring within the control-flow, for example, specific event orders. We, to some degree, reduce the amount of data that can possibly influence the behavior and thus make concessions regarding the analysis precision in order to enable practical feasibility. The actual abstraction degree is implementation-specific. We always require logical soundness for our abstractions though, but not necessarily logical completeness. As a result, we may get false positives in our analysis results. To what degree these abstractions influence the analysis results is discussed in Chapter 5.

Note that the test case model is only concerned with the abstract test case specification. For instance, in testing with TTCN-3, the involved artifacts are the abstract test specification, as well as an adaption layer that, for example, maps abstract messages to concrete messages and sends these concrete messages through a non-abstract medium. Such adaption layers as found in the TTCN-3 architecture, do not play a role for the analyses presented in this chapter, i.e., we only deal with the analysis of the abstract artifact. Therefore, possible anomalies in the adaption layer cannot be analyzed using the methods presented in this thesis.

In comparison to a static grammatical model of the syntax, i.e., a metamodel representing the language structure, also called the abstract syntax of the language [128], our test case model is also different. The test case model represents the actual behavior of the analyzed behavior. However, from an abstract syntax model and possibly a corresponding binding data structure, we cannot directly derive what the overall behavior looks like, i.e., including the behavior that is included due to function calls, for instance. To achieve that, we have to interpret and follow the paths and references of the abstract syntax, which essentially equates to a special-case execution of the behavior. Exactly this kind of special-case execution (or simulation) over the abstract syntax of the analyzed language is what we perform in order to gather the relevant log data.

4.3. Model Reconstruction

When we analyze test specifications, we have a test specification available in a concrete language, such as TTCN-3. These test specifications are then the subject of the analysis. However, they do not have the form of a model as presented in Section 2.2. For that purpose,

a method is necessary that reconstructs (or reverse engineers) the internal model of the test specification using the formal model that we have in our mind—in our case, an EMIOTS model. In general, there are two approaches to obtain such a behavioral model:

- Abstractly simulating the behavioral control-flow and data-flow and using this internal information to build the model (model reconstruction by abstract interpretation).
- Monitoring the actual execution of instantiated test specifications and using logged information from these executions to reconstruct the model (model reconstruction from observations).

The challenge of the first approach is the fact that it must respect the operational semantics of the language under investigation to some degree. In essence, such a method can be regarded as special-case interpretation. The second approach on the other hand requires an actual execution of the test specification under analysis. The observations necessary to build the model are either inferred from the executions, or the test specification is instrumented to log the necessary information. Here, we can rely on existing compilers for instantiated test specifications and as a result, it is only necessary to handle the language semantics to a limited degree on the user level. We can assume that existing compilers implement the language semantics correctly.

For both methods, the completeness of the reconstructed model depends on the amount of distinct traces gathered and which coverage criteria they represent. When system models (as opposed to test models in our case) are reconstructed from observations, the logs are often collected when the system is responding to stimuli of black-box tests. This method is based on the assumption that the tests provide some sufficient means of behavioral coverage (such as branch coverage) of the system under analysis. Naturally, this approach is not possible or feasible when a test behavior model should be reconstructed: there are no tests that can be executed against the test behavior under analysis. An execution of tests against a real SUT is not be sufficient: when a test case passes, we do not have the failure case covered. Likewise, we do not have the passing behavior covered if a test case fails. Other behavior might not be covered due to external configurations or because additional error behavior (making the test behavior more robust) did not happen. Running test cases against a system can generally never represent all behavioral possibilities of a single test case. Furthermore, we presume that the actual SUT is not available for the model reconstruction as our goal is the verification of properties prior to the execution against a real SUT.

To solve the problem that an SUT is not available and that a specific amount of behavior must be covered for a model reconstruction, we present a model reconstruction method based on abstract interpretation. This method is essentially performed in two steps:

- The test case under analysis is abstractly interpreted (or simulated) until a coverage criterion is met. During these simulations, events of a specific format are logged.

- The logged events are passed to the model reconstruction algorithm that incrementally builds a model from these sets of events. When the simulator is finished, our model is complete.

In the following, we present these two steps. First, we present the information that needs to be logged. Afterwards, we demonstrate how to use this information to reconstruct models.

4.3.1. Logging

The first vital part of our model reconstruction method is determining exactly what information and events should be traced or logged[5] to reconstruct a behavioral EMIOTS model. For that purpose, we define a log tuple ρ that captures event information, i.e., it represents a single observation. One simulation captures a sequence of log tuples $\rho := \rho_0, \rho_1, \ldots, \rho_m$. The actual reconstruction is performed over a sequence of log sequences $\lambda_0, \lambda_1, \ldots, \lambda_n$. Each log sequence λ_i adds additional information to the reconstructed model and refines it.

On the one hand, the EMIOTS model of one process or component represents the interprocedural control-flow of its sequential behavior. On the other hand, the set of variables and the update actions that manipulate variables represent the data-flow. We intuitively need to log:

- Structural information.
- The communicating events.
- Information about the data-flow.

For the construction of the interprocedural behavioral control-flow, the log must contain branch events for decision points, such as if-statements or loop-conditions, as well as corresponding join (or meet) events where multiple branched paths of behavior coincide. Essentially, these are often points within the behavior where new scopes are introduced and closed. Unique event identifiers are used to distinguish every scope start and scope end event. In addition, a scope event can be marked as an event that prematurely ends a recorded complete scope stack and where no further join events are expected. An example for such a case is a *return* statement, as found in many languages.

The second kind of events that are recorded are communicating events, i.e., input actions and output actions. They are used to express dependence between multiple test processes and are decisive for where the behavior of these components is synchronized. Also, they can be used for message-flow based verifications, for example, for checking certain message orders.

[5] In the remainder, we use the terms logs and logging instead of traces and tracing in order to avoid misunderstandings with Definition 2.4.

Given the formal definition of the log sequences above, we define the tuple ρ as one event in a sequence of events that represent one path of the test behavior.

$$\rho := (pid, id, ss, se, l, i, o, d, pn, pv). \tag{4.3.1}$$

Each undefined tuple element is denoted by an ε. The respective tuple elements are defined as follows:

- $pid \in \mathbb{N} \cup \{\varepsilon\}$ refers to a process identifier on which the event happened,
- $id \in \mathbb{N} \cup \{\varepsilon\}$ refers to an event identifier local to the pid,
- $ss \in \{\varepsilon, 1\}$ is a flag indicates the start of a new scope if $ss = 1$,
- $se \in \{\varepsilon, 1, 2\}$ refers to an event that indicates the end of a scope if $se = 1$ and a premature scope stack closure if $se = 2$,
- $l \in \{\varepsilon, 1\}$ is a flag that indicates whether an event is an actual event ($l = \varepsilon$) or a look-ahead event ($l = 1$),
- i refers to a message input event (p, m) including a target port p and a message label m. We also use the notation $p?m$ to describe such an event,
- o refers to a message output event (p, m) including a target port p and a message label m. We also use the notation $p!m$ to describe such an event,
- d denotes an event description string,
- pn refers to the name of a data identifier (data name),
- $pv \in \{\varepsilon, 0, 1\}$ refers to the boolean value of a data value corresponding to pn.

The tuple represents multiple different types of events. We effectively deal with six different concrete event types. The event type of an event ρ can be identified by the $etype$ function that identifies the message type by case differentiation of consistency rules. For better readability, we assign symbolic names and shortcuts as results of the $etype$ function.

- $etype(\rho) =$ scopeStartEvent (ss),
- $etype(\rho) =$ scopeEndEvent (se),
- $etype(\rho) =$ messageInputEvent (mie),
- $etype(\rho) =$ messageOutputEvent (moe),
- $etype(\rho) =$ dataEvent (de),
- $etype(\rho) =$ internalEvent (ie), and
- $etype(\rho) =$ invalidEvent

Table 4.1 lists the consistency rules for each tuple field that need to be fulfilled for certain event types. The consistency rules can be used to identify a certain event and to validate whether an event tuple is consistent. For example, if we receive an event $(3, \varepsilon, 1, \varepsilon, \varepsilon, \varepsilon, \varepsilon, \varepsilon, \varepsilon, \varepsilon)$, it matches a *scopeStartEvent*. That way, all six different event types are encoded in the ρ tuple.

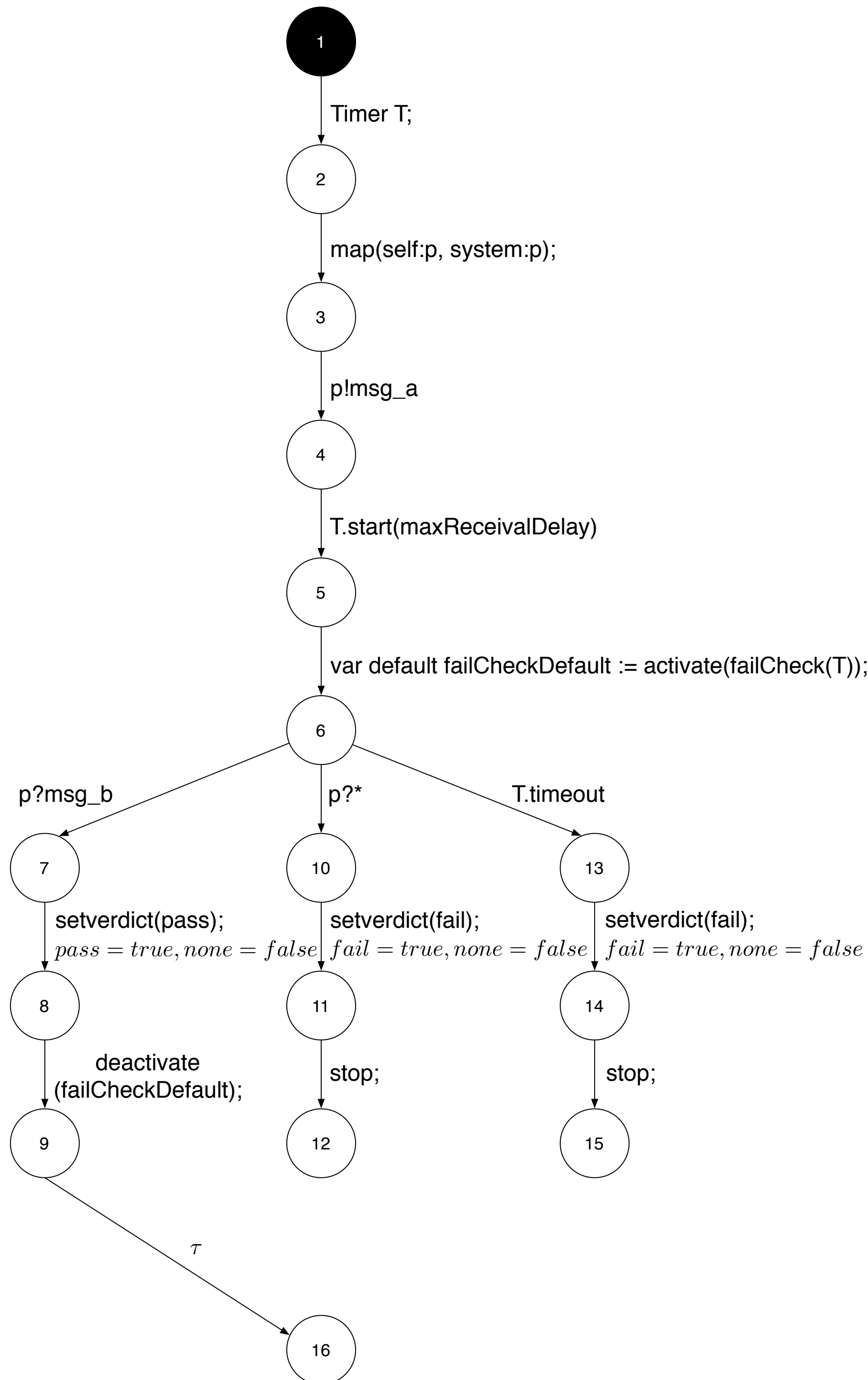


Figure 4.4.: Example: Mapping Log Events to a Model

$etype(\rho)$	**pid**	**id**	**ss**	**se**	**l**	**i**	**o**	**d**	**pn**	**pv**
scopeStartEvent	$\neq \varepsilon$	ε	1	ε	ε	ε	ε	ε	ε	ε
scopeEndEvent	$\neq \varepsilon$	ε	ε	$1 \vee 2$	ε	ε	ε	ε	ε	ε
messageInputEvent	$\neq \varepsilon$	$\neq \varepsilon$	ε	ε	$\varepsilon \vee 1$	$\neq \varepsilon$	ε	ε	ε	ε
messageOutputEvent	$\neq \varepsilon$	$\neq \varepsilon$	ε	ε	$\varepsilon \vee 1$	ε	$\neq \varepsilon$	ε	ε	ε
dataEvent	$\neq \varepsilon$	$\neq \varepsilon$	ε	ε	$\varepsilon \vee 1$	ε	ε	ε	$\neq \varepsilon$	$\neq \varepsilon$
internalEvent	$\neq \varepsilon$	$\neq \varepsilon$	ε	ε	$\varepsilon \vee 1$	$\neq \varepsilon$	ε	$\neq \varepsilon$	ε	ε
invalidEvent	Otherwise									

Table 4.1.: Log Consistency Rules

The scope start event and the scope end event are used to log structural information for the control-flow, for example, when a scope begins and when it ends. The message input event and the message output event are used to log message-passing events between the test processes. The data event is used to reconstruct the data-flow of boolean variables regarding the data values of interest. Finally, the internal event is used to identify basic blocks in the control-flow graph, i.e., behavioral pieces without any jumps or jump targets where no communicating or data events occur. Their description can be found in the d-element of the logging tuple.

All events, except for the scope start event and the scope end event, may be flagged as look-ahead events. Multiple subsequent look-ahead events announce a set of events that may occur next, but only one actually occurs within a single test run. This information allows the model reconstruction algorithm to see different routes through the behavior in advance allowing it to take action at the decision point if necessary.

Table 4.2 depicts an example log for a single test run λ_i. The example log essentially represents what could become the leftmost path of Figure 4.4. The events $\rho_0, \ldots, \rho_3$ are initialization events where the data entities are assigned an initial value. Events $\rho_9, \ldots, \rho_{11}$ are look-ahead events which announce possible next events. The event ρ_{13} (as opposed to $\rho_9, \ldots, \rho_{11}$ actually takes place and it corresponds to ρ_9.

4.3.2. Behavior Model Reconstruction

The prerequisite or input for the model reconstruction algorithm is a sequence of logs $\lambda_0, \lambda_1, \ldots, \lambda_n$ each with events $\rho_i, 0 < i < m$ where all n logs representing executions are expected to represent a complete branch coverage of the test specification under analysis. Each log $\lambda_j, 0 < j < n$ is processed successively, where each iteration of the algorithm results in an increment of a partial model. Missing parts of the model are added successively. The model reconstruction is exact regarding the information the logs provide, i.e., each event has its clear and unambiguous place in the model. Due to the recording of look-ahead events, the algorithm offers the possibility to take action towards the concerned test com-

event	**etype**	**pid**	**id**	**ss**	**se**	**l**	**i**	**o**	**d**	**pn**	**pv**
ρ_0	de	1	1	ε	ε	ε	ε	ε	ε	pass	0
ρ_1	de	1	2	ε	ε	ε	ε	ε	ε	fail	0
ρ_2	de	1	3	ε	ε	ε	ε	ε	ε	inconc	0
ρ_3	de	1	4	ε	ε	ε	ε	ε	ε	none	1
ρ_4	ie	1	5	ε	ε	ε	ε	ε	Timer T;	ε	ε
ρ_5	ie	1	6	ε	ε	ε	ε	ε	map (self:p, system:p);	ε	ε
ρ_6	moe	1	7	ε	ε	ε	ε	$p!msg_a$	ε	ε	ε
ρ_7	ie	1	8	ε	ε	ε	ε	ε	T.start (maxRe-ceivalDe-lay);	ε	ε
ρ_8	ie	1	9	ε	ε	ε	ε	ε	var default failCheck-Default := activate(failCheck(T));	ε	ε
ρ_9	mie	1	10	ε	ε	1	p?msg_b	ε	ε	ε	ε
ρ_{10}	mie	1	11	ε	ε	1	p?*	ε	ε	ε	ε
ρ_{11}	ie	1	12	ε	ε	1	ε	ε	T.timeout	ε	ε
ρ_{12}	ss	1	ε	1	ε	ε	ε	ε	ε	ε	ε
ρ_{13}	moe	1	10	ε	ε	ε	p?msg_b	ε	ε	ε	ε
ρ_{14}	ie	1	13	ε	ε	ε	ε	ε	setverdict (pass);	ε	ε
ρ_{15}	de	1	14	ε	ε	ε	ε	ε	ε	pass	1
ρ_{16}	de	1	15	ε	ε	ε	ε	ε	ε	none	0
ρ_{17}	ie	1	16	ε	ε	ε	ε	ε	deactivate (failCheck-Default);	ε	ε
ρ_{18}	se	1	ε	ε	1	ε	ε	ε	ε	ε	ε

Table 4.2.: Example: a Test Run Log λ

ponent at each decision point, for example, to steer the behavior of the test component in some way. This means, we can not only create the partial model from the behavior that has been executed and recorded in the log, but we may also use look-ahead events for making decisions in the simulation process.

The formal description of the model reconstruction algorithm can be found in Appendix A.1. An example in Appendix A.2 demonstrates how the algorithm is applied. In the following, we informally present the basic idea of the algorithm and refer to the appendix for more details. The intuition behind the model reconstruction algorithm is the conversion of events ρ_i of an execution log λ_j into actions of the EMIOTS model and to insert states before and after each action. By using a combination of the scope ids, the scope id history, and event ids, we are able to identify unique positions within the overall control-flow of the behavior. By using this information, we are able to identify equal transitions and, therefore, match them to existing transitions in our model. That way, we add states and transitions that are not already part of the model and match these transitions that exist already. Depending on the event type (and thus the information that the event carries), the model reconstruction algorithm behaves differently and updates its internal state. Therefore, we differentiate the algorithm behavior for each event type:

- **Scope Start Event:** The scope id is pushed on a scope stack that is used to track the scope history (from this stack, we create unique event identifiers).
- **Scope End Event:** The scope id is popped from the scope stack. The new state position in the target EMIOTS is determined. Either it is identified by matching the scope history after the scope stack has been popped or a new state is introduced if no match is found.
- **Message Input/Output Event:** If the message input or output event can be matched against the current model, we proceed to the next state that has been identified as the target state of the matching transition. If no match is found, a new state and transition are created for the event and data structures are updated to match the new transitions when queried. We move forward to the new state or the matched target state if the event was not a look-ahead event.
- **Data Event:** The overall behavior is comparable to the message input/output event. However, the data event carries information about data and its manipulation. Therefore, the actions are variable-updating actions in the set A_U.
- **Internal Event:** Like a data event, but no data manipulation takes place and the actions are part of the set A_N.

In a certain way, the behavior of the message input/output event, the data event and the internal events is somewhat similar. Thus, the algorithm seems like a straightforward event-matching algorithm, and in fact, the model reconstruction algorithm does not contain a lot of complexity or logic. However, there are a various fine-grained details of the algorithm that are not immediately apparent from the simplified description above—especially regarding

the realization of the matching. Therefore, we refer the reader to Appendix A.1 for these details.

The model reconstruction algorithm itself only performs constant time $O(1)$ operations to process the event log. Computational complexity is introduced by the test case simulation or, in other words, by the amount of event logs that need to be created and processed to achieve the desired coverage criterion.

Our criterion to create complete models of the behavior under analysis is that in each set of event logs, every behavioral branch must be covered at least once. Ensuring that every branch is covered means that every branch in the original behavior has a corresponding branch in our model. Similarly, as we model only variables and their manipulations as opposed to concrete variable values, it is sufficient to cover the branches due to the fact that once we visited every branch, we have seen all variable manipulations that are reachable within the behavior. The only behavior that we do not be able to find in our models is behavior that is unreachable from our start state. Due to these facts, branch coverage of the test behavior that we intend to reverse engineer is sufficient. Increasing the coverage criteria does not yield any additional benefit in this case.

4.3.3. Test Case Simulation

The previous two subsections explained the kind of information we need to log and the use of this log information to reconstruct a behavioral model for a sequence of logs $\lambda_0, \lambda_1, \ldots, \lambda_n$. In a testing environment, tests are executed against an SUT. In order to analyze quality aspects of the test suite, however, we do not want to depend on the existence of an SUT. Instead, we want to analyze test cases before an actual execution against the SUT. Therefore, we perform an abstract interpretation over the test case specification. This abstract interpretation is driven by our incrementally reconstructed partial models.

Figure 4.5 depicts how the abstract interpretation works in the context of the model reconstruction. The abstract interpretation is performed in the simulated tester component. This simulated tester executes the abstract test specification. From this execution, event logs are produced and then processed by the model reconstruction algorithm.

In fact, the abstract interpretation of the test behavior executes the test behavior, but only in the way necessary to gather the required log data. For this purpose, the abstract interpretation does not evaluate variables or data in general, but merely chooses different paths in the control-flow to cover as many branches of the behavior as possible and thereby also neglecting whether the chosen execution is a valid execution of the test behavior in terms of its data-flow. The strategy is to visit as many different branches of each test component behavior as quickly as possible. Other properties, such as port mappings and connections are established just like in a normal test case execution. Actual communication among the test components or between the test components and the SUT does not take place. Parallel properties such as synchronization points are evaluated later-on in the model checking stage. Hence, the method allows a special-case execution of the abstract test specification

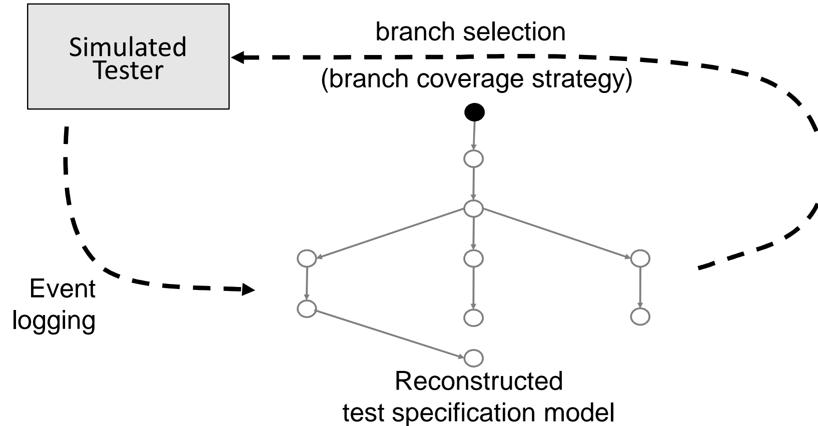


Figure 4.5.: Test Behavior Simulation

without the appropriate test environment. The execution retains some properties (such as port connections or default activations in TTCN-3) of a real test case execution while neglecting others (such as data-flow based conditions). It is tailored for the production of the necessary log events that allow an accurate reconstruction of the test case model.

The model reconstruction algorithm creates a partial model increment from the information received from the simulated tester. In this model, every state that is executed in the abstract interpretation is colored with color c_1. When the abstract interpretation detects a branch where a choice can be made within the behavior, it chooses transitions to not yet colored nodes first, transitions to c_2 colored nodes second, or chooses an arbitrary transition otherwise. After the partial model increment and the abstract interpretation iteration is finished, a second coloring is performed on the partial model: starting from the last reached state, we go backwards in the model choosing backward transitions which have the c_1 color. For every transition we go backwards, we color the reversed state with color c_3. We continue backward traversal until we reach a state that has children that are not yet colored with c_3 or until we reach the start state. Afterwards, we recolor all states with c_1 or c_2. In other words, the c_1 color marks the nodes of a current execution, c_3 marks branch nodes that have been visited already and c_2 is the overall color for nodes that have been visited at some point. This whole coloring procedure makes sure that in each iteration of the abstract interpretation, we choose new behaviors that we have not yet covered in our model. A detailed description of the coloring algorithms can be found in Appendix A.3.

The backwards coloring after an iteration essentially amounts to complexity $O(n)$ with n being the length of the execution trace. To actually achieve branch coverage for the complete model reconstruction, we would have, in the worst case, one simulation iteration for each branch in the model. Therefore, the total computational complexity of the model reconstruction and the simulation together is linear with respect to the amount of branches

in the test behavior. To some degree, the problem has similarities to a classic backtracking algorithm, as we essentially explore the model like maze. However, each choice taken within the algorithm actually amounts to a part of the solution. Therefore, as opposed to backtracking, we make progress in each iteration.

4.4. Test Case Analysis

Following the definition of our formal model and the description of the model reconstruction method which we use to reconstruct our test models from test specifications, we can finally deal with the actual application or purpose of the effort we have put in creating the model. While the model is certainly useful for understanding and documenting the test behavior as well, our primary goal is to use the model to check specific properties on it. These properties are mostly generic descriptions of what should and should not happen in the test behavior. In the context of testing, for example, we expect that at some point the test makes a decision about the test verdict. Here, we deal especially with properties that cannot be analyzed using static analysis and where a dynamic analysis provides an advantage (e.g., the dynamic analysis may provide detailed execution trace of the detected problem). For the sake of simplicity and to differentiate these properties clearly from the ones that can be analyzed statically, we refer to them as *dynamic properties*.

The properties that we present are based on the EMIOTS model representing abstract test cases. We deal especially with issues that are caused by human mistakes, i.e., anomalies that occur when the test developer (of manually written test cases) does not pay attention to some particular detail. We do not attempt to cover all kinds of imaginable dynamic anomalies, but the ones that are related to specific orderings of certain events in the behavior and that are behavior-locking. We differentiate between generic properties that can be detected in models of communicating systems, properties that can be analyzed in test models, i.e., models that specifically represent test cases, and finally we present properties that are specific to the TTCN-3 test specification language. We present each anomaly in a fixed format. This fixed format consists of a *name*, a *description*, an *example*, and an *analysis* description about the necessary variables, as well as a generic LTL formula using these variables where applicable. In the following sections, we present four test-specific anomalies, four TTCN-3-specific anomalies, and five generic anomalies, i.e., a total of 13 anomalies. In each analysis description that can be achieved using property descriptions in LTL, we refer to the nature of each possible temporal logic formula, i.e., whether it describes correct behavior or error behavior. Furthermore, the variables for modeling events refer to their true value when used in temporal logic formulas. While the example models are supposed to be generic and not language-specific, the actions syntactically refer mostly to TTCN-3 statements to make the examples easier to understand. Note that the example models only represent possible excerpts of a test case for demonstrational purposes. Note: in some anomaly descriptions, we will refer to the concept of an *interruption event*. An interrupt event is an event that resets the value of a variable right before it is set. For example, when a variable value is true and

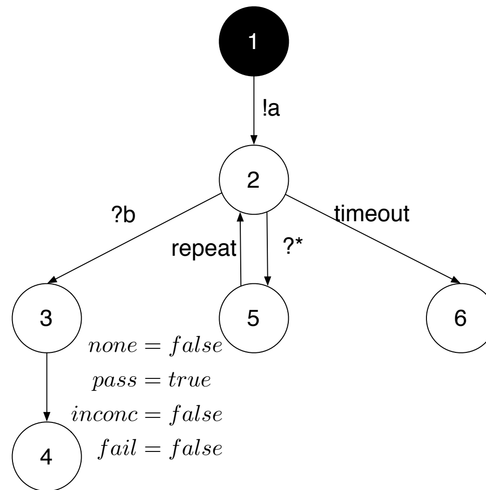


Figure 4.6.: Example: Missing Test Verdict

in some follow-up transition is again set to true, we have to insert such an interruption event to actually notice this subsequent value change to the same value. We will refer to these events when they become necessary.

4.4.1. Test-Specific Anomalies

In the following sections, we present test-specific anomalies, i.e., anomalies that somehow include concepts that are specific to test specifications. We primarily deal with anomalies that are related to test verdicts.

4.4.1.1. Missing Test Verdict

Description: In the test model, there may be a path in which no verdict is set. Such a path is likely to be unintentional as a test is always supposed to end in a verdict, either pass, fail, or possibly inconclusive. An unset verdict indicates that a specific behavioral path remained unconsidered by the test developer and that the path does not have a sensible result.

Example: The model in Figure 4.6 depicts a message passing test case behavior. In this test case, a stimulus $!a$ is sent to the SUT. The reaction to this stimulus in state 2 can be threefold: either a message $?b$ is received from the SUT and the verdict is set to pass, any other message is received, the test case ignores the message and returns to the state 2, or timeout happens.

The verdicts are modeled using four boolean variables each of which represents the state of a possible verdict, i.e., there are variables *pass*, $fail$, *inconc* (for inconclusive), and *none*

for an unset verdict. The variables *pass*, *fail*, *inconc* are initialized with a false value. The variable *none* is intialized with a true value.

In fact, only the first alternative is actually dealing with the verdict. The timeout case is particularly critical as it provides an exit point from the test case behavior, but does not set a test verdict.

Analysis: We can express an LTL formula (describing correct behavior) that checks whether the verdict is initialized with no verdict and whether the verdict will change at some point to either *pass*, *inconclusive*, or *fail*. The following formula makes sure that there is no path where the verdict is explicitly unhandled:

$$\phi := (\neg pass \wedge \neg inconc \wedge \neg fail)\ \mathcal{U}\ (pass \vee inconc \vee fail) \tag{4.4.1}$$

4.4.1.2. Fail/Inconc Verdict Decision Before Communication

Description: In manually written tests, it is sometimes practice to initialize the verdict of a test case to pass prior to any behavior—especially communicating behavior or behavior requiring user interaction (such as the action statement in TTCN-3)—that takes place in tests. The idea is that the verdict is then only overwritten when erroneous or inconclusive behavior occurs and thus the missing verdict anomaly can be avoided. A similar string of arguments can be used to set default verdicts to a failing or inconclusive verdict: rather than having a test case pass unexpectedly, it might be sensible to let a test case fail in case of doubt (i.e., human mistakes for example) since in that case, the results would be checked, whereas a pass verdict commonly does not cause any action.

However, depending on the language or test framework used for describing the test specification, this idea might be bad: TTCN-3, for example, does not allow to overwrite fail or inconclusive verdicts with pass verdicts. Therefore, such default initializations to non-pass verdicts prior to any communicating behavior should be avoided in all situations.

Example: Figure 4.7 illustrates a variant of Figure 4.6 where a default fail verdict (the transition from state 1 to state 2) is set prior to any communication taking place (i.e., before the $!a$ stimulus is sent to the SUT). We model the first point of communication with an additional boolean variable *comm* which is changed from false to true on the initial communicating events.

Analysis: The following formula (describing correct behavior) ensures that a communicating event takes places prior to the first change to fail or inconclusive verdicts:

$$\phi := (none \wedge \neg\ pass \wedge \neg\ inconc \wedge \neg\ fail)\ \mathcal{U}\ comm \tag{4.4.2}$$

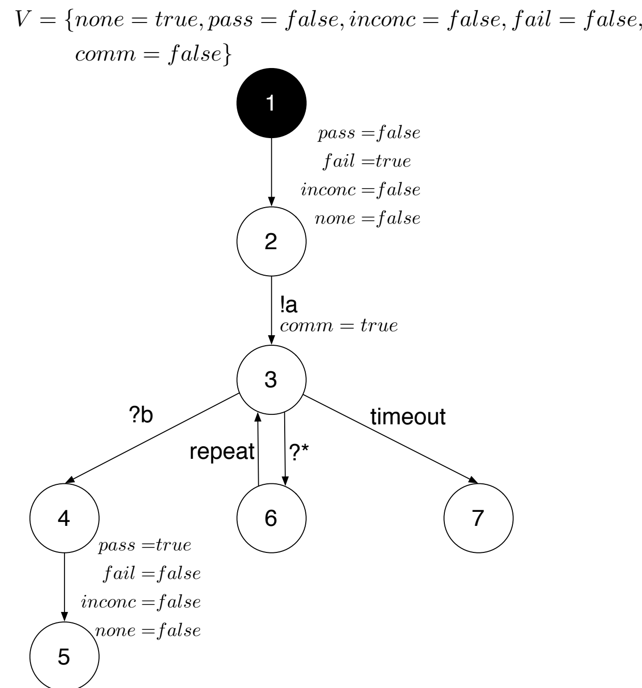


Figure 4.7.: Example: Fail/Inconc Verdict Decision Before Communication

4.4.1.3. Illegal Verdict Overwrite

Description: When a fail or inconclusive verdict is set in a test, the decision is supposed to be immutable. The reason is that once faulty behavior or data is diagnosed, changing this fault diagnosis to a pass verdict is irrational: there is nothing that can repair wrong behavior or data once it is diagnosed. Chances are that setting a pass verdict after a fail or inconclusive verdict conceals the previous fault diagnosis and hides it. For that reason, languages such as TTCN-3 prohibit overriding inconclusive or fault verdicts. This, however, is not always be the case. With this anomaly, we describe the case where the test framework or language semantics do not catch this kind of misbehavior.

Example: For this property, we reuse the example from Figure 4.7. In this example, a verdict is not only set prior to any communication, but it is also initialized to fail (transition from state 1 to state 2) and may be overwritten in the transition from state 4 to state 5. In this case, the left-most path would contain an invalid illegal verdict overwrite. Whether or when communication takes place for the first time does not play a role for this anomaly. Therefore, the *comm* variable is not necessary for the analysis.

$V = \{none = true, pass = false, inconc = false, fail = false\}$

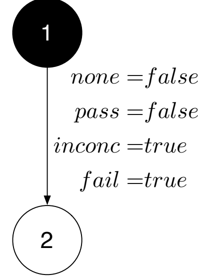


Figure 4.8.: Example: Verdict Consistency

Analysis: The formula (describing correct behavior) essentially requires that a fail verdict cannot be overwritten and that an inconclusive verdict can only be overwritten by a fail verdict.

$$\phi := \Box\ ((fail\ \rightarrow\ \Box\ fail)\ \wedge\ (inconc\ \rightarrow\ (\Box\ inconc \vee (inconc\ \mathcal{U}\ \Box\ fail)))) \tag{4.4.3}$$

4.4.1.4. Verdict Consistency

Description: Test verdicts in EMIOTS models as presented in the anomalies before are modeled using four variables, where the variables each models none, pass, inconclusive, and fail verdicts. A test verdict in such a model is consistent only when one of these four variables is true exclusively, i.e., there may not be a state where the value of more than one variable is true. Such situations can nevertheless occur, for example, if the model construction is faulty.

Example: The example shown in Figure 4.8 is a straightforward inconsistency: an EMIOTS model with two states and one transition where the variables are directly altered in an inconsistent way in this single transition. Here, two variables are changed to the value true, where only one is consistent.

Analysis: Model consistency regarding verdicts can be verified with the following formula describing correct behavior:

$$\begin{aligned} \phi := \Box\ (&(none \wedge \neg pass \wedge \neg inconc \wedge \neg fail) \vee \\ &(\neg none \wedge pass \wedge \neg inconc \wedge \neg fail) \vee \\ &(\neg none \wedge \neg pass \wedge inconc \wedge \neg fail) \vee \\ &(\neg none \wedge \neg pass \wedge \neg inconc \wedge fail)) \end{aligned} \tag{4.4.4}$$

4.4.2. TTCN-3 Specific Anomalies

In this section, we present anomalies that refer to concrete features and language constructs of TTCN-3. In some cases, the anomalies refer back to Section 4.4.3 where the TTCN-3 anomalies are instances of a generic anomaly or an anomaly pattern.

4.4.2.1. Idle PTC

Description: In TTCN-3, the action to create and start a test component with parallel test behavior is separated. As a result, there is the possibility that a test component is created, but never started. Such a component is called an *Idle PTC* [18] and is a component that, albeit created, never contributes to the test verdict of the test that is executed since no test behavior is ever executed on it. In this scenario, the test developer either never intended the PTC to run in the first place and the concerned code pieces can be regarded as clutter that can be removed, or the developer actually intended to run this parallel test component, but missed or forgot to start it. More generally, this smell can be regarded as *Event Pair Asymmetry* (see Section 4.4.3).

Example: Figure 4.9 shows an example model where the second event pair, i.e., the start component event is missing. This means that a component is created, but not started. Here, the variable *event_a* represents the occurrence of the component creation, whereas *event_b* represents the occurrence of the corresponding start component event. Each statement changes to true when the corresponding statement is executed. Thus, only the *event_a* variable is actually manipulated within the model and the second event pair is missing.

Analysis: As stated before, the analysis is an instance of the *Event Pair Asymmetry* anomaly analysis. The formula (describing correct behavior) is as follows:

$$\phi := \Box\,(event_a \;\rightarrow\; \Diamond\; event_b) \tag{4.4.5}$$

4.4.2.2. Default Asymmetry

Description: Default altsteps can easily cause unexpected behavior when test developers do not handle them with the required consistency. Activated defaults which have not been deactivated, not only stay active until the end of the scope unit, but they stay active until the corresponding test case terminates. In addition, their functionality is not limited to the activation scope level, but activated defaults are executed in any alt-statement or altstep following their activation until they are deactivated. This includes altsteps in dependent function calls and similar. As a result, defaults should always be deactivated as soon as they are not needed or wanted any more.

Missing deactivate statements possibly conceal where exactly the default behavior is attached to in which case the test developer may misinterpret the test code at hand. The test

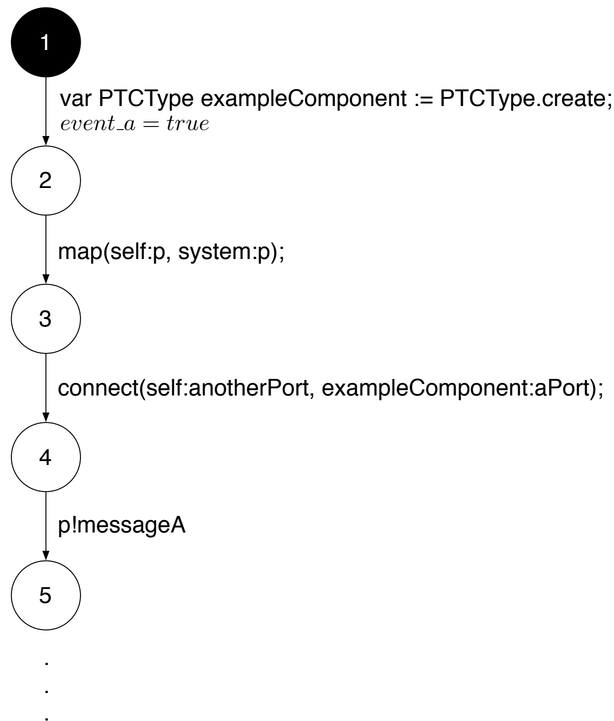


Figure 4.9.: Example: Idle PTC

code becomes hard to understand and the developer may not even notice the difficulties that are possibly involved.

Example: In fact, the altstep asymmetry is also an *Event Pair Asymmetry* like the *Idle PTC* anomaly. Here, we have two events *activation* and *deactivation* where *deactivation* is required to occur at some point after *activation* took place. Figure 4.10 illustrates the analogous behavior. The figure shows that a corresponding deactivate statement is missing.

Analysis: The altstep asymmetry is an instance of the *Event Pair Asymmetry* anomaly analysis. The formula (describing correct behavior) is as follows:

$$\phi := \Box\,(activation\ \rightarrow\ \Diamond\ deactivation) \tag{4.4.6}$$

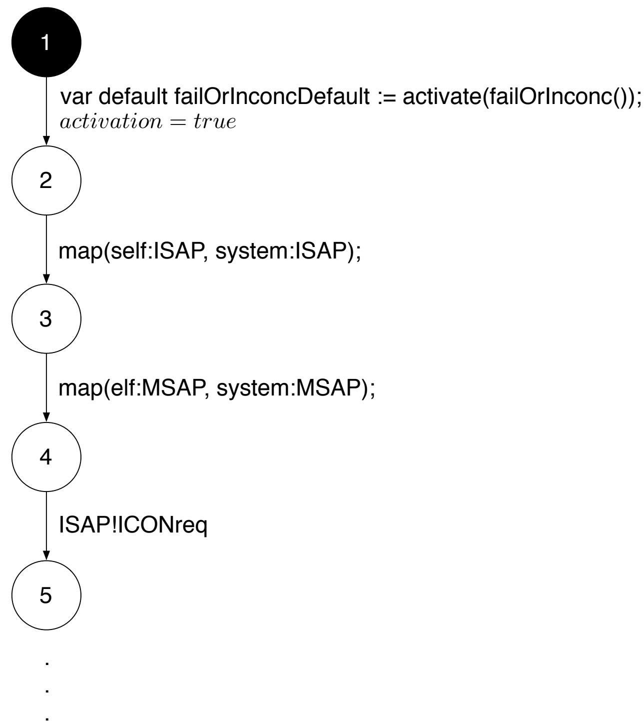


Figure 4.10.: Example: Altstep Asymmetry

4.4.2.3. Send/Receive on Unconnected/Unmapped Ports

Description: Ports of components must be connected either to the TSI or to ports of PTCs before send or receive operations can take place. Therefore, send and receive statements are illegal if no proper connections have been established prior to the communication.

In this anomaly, we analyze the relationship between the connection of a port and its use. We model this problem with two variables: one variable *pConnected* describes the event when a port *p* of a component is connected, whereas the other variable *pMessage* describes whether a message is sent or received through port *p*. At each send or receive statement where this port is referenced, we need to assert that the variable representing the referenced port has a true value, i.e., it is connected or mapped.

Example: Figure 4.11 illustrates an example of the anomaly. There are two paths in the model that lead to the message $p!message$ being sent on the transition between states 4 and 5. The left-hand path does not connect or map the port *p*, whereas the right-hand path of the model does map port *p*. Thus, the left-hand path of the model represents a path with an anomaly as a message is sent through *p* when *p* is not yet connected.

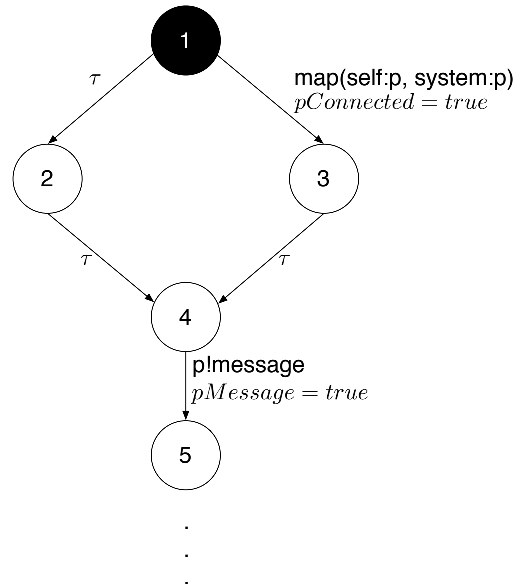


Figure 4.11.: Example: Send/Receive on Unconnected/Unmapped Ports

Analysis: The analysis can then be expressed in LTL as follows (describing correct behavior):

$$\phi := \square(pMessage \rightarrow pConnected) \tag{4.4.7}$$

The formula ensures that whenever a message is sent, the *pConnected* variable must be set to true, thus ensuring that the port under analysis is connected or mapped. Note: interruption events must be set for each *pMessage* event.

4.4.2.4. Send/Receive on Stopped/Halted Ports

Description: TTCN-3 port operations such as *send* and *receive* are only allowed when the port is listening. When a component is started, the *start* command is executed implicitly and the ports can be used for *send* and *receive* operations. However, the *halt* and *stop* commands on a port disallow any further communication on it. Hence, communicating operations on a stopped or halted port lead to behavior that is either badly designed or faulty.

Example: The anomaly is a variant of the *Send/Receive on Unconnected/Unmapped Ports* anomaly. Instead of a variable *pConnected* as illustrated in Figure 4.11, we imagine

a variable $pHalted$ (where a corresponding transition would halt a port instead). Unlike in the situation of unconnected and unmapped ports, the correct behavior is a false value for $pHalted$ instead of true, i.e., the communication operations $pMessage$ are valid only if $pHalted$ is false instead of true. The initial value of the $pHalted$ variable is $false$.

Analysis: The analysis can then be expressed in LTL as follows (describing correct behavior):

$$\phi := \Box(pMessage \rightarrow\ !pHalted) \tag{4.4.8}$$

The formula is essentially the same formula as the formula for the *Send/Receive on Unconnected/Unmapped Ports* anomaly. However, $pHalted$ and $pConnected$ are switched and $pHalted$ is negated. As before, subsequent $pMessage$ events need interruption events.

4.4.3. Generic Anomalies

The subsequent anomaly descriptions are generic, i.e., they refer to anomalies that may be applicable in any software system that is constructed as a model. The anomalies are not specific to the fact that we deal with tests. In fact, some of these anomalies are well-known (for example, data-flow anomalies), but they—to our knowledge—have not yet been described in the cataloged manner using LTL specification patterns (where applicable). In this regard, this section about generic anomalies is of common interest for anyone dealing with quality assurance of software systems and not just testing experts.

4.4.3.1. Data-Flow Anomalies

Description: Data-flow anomalies [63] are issues that are related to a sequence of variable definition (d), undefinition (u) and referencing (r) operations:

- A variable is referenced before its definition (ur).
- A defined variable is redefined before it is referenced (dd).
- A variable is defined and gets undefined before it is referenced (du).

Such operation sequences can identify unintentional, problematic, or even erroneous code. While it is possible to statically identify whether a data-flow anomaly might happen, a dynamic analysis is needed to pinpoint when and where exactly a data-flow anomaly might occur.

Example: Figure 4.12 illustrates a model that contains dd and du anomalies. We represent each variable in the test specification with three different boolean event variables in the EMIOTS model, where one event variable d represents the definition, one event variable u

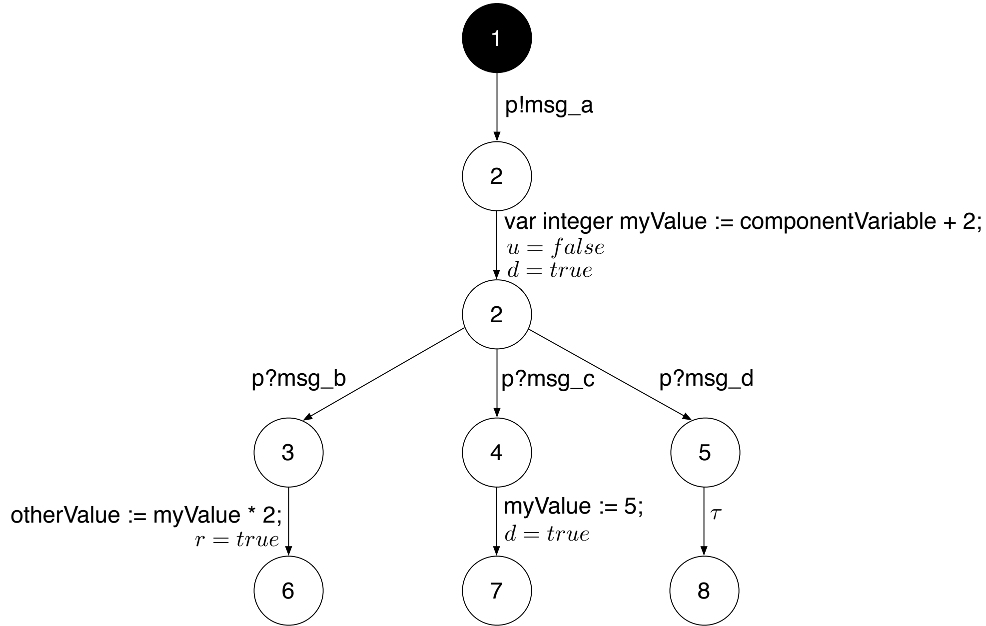


Figure 4.12.: Example: Data-Flow Anomalies

represents the undefined state of the variable, and the event variable *r* represents the referencing of the variable. In the example, the three variables represent the integer variable *myValue*.

In the example, a stimulus *p*!*msg_a* is sent to the SUT. Following this stimulus, the integer variable *myValue* is initialized. As a response to the stimulus, the test case expects either *p*?*msg_b*, *p*?*msg_c*, or *p*?*msg_d*. In the first case, the variable *myValue* is referenced for another statement. In the second case, *myValue* is redefined although it has not been referenced before (dd anomaly). In the third case, only the message *p*?*msg_d* is received, but the variable is neither redefined, or referenced. Therefore, *myValue* is not needed for this case (du anomaly). Therefore, we have data-flow anomalies when either a timeout happens or when *p*?*msg_c* arrives.

Analysis: The following formula (describing correct behavior) ensures that the ur anomaly does not occur in the desired behavior. The formula expresses this by demanding that if a variable is referenced, the undefined variable must be defined and then referenced.

$$\phi := \Box(r \rightarrow ((u \land \neg d \land \neg r)\ \mathcal{U}\ (\neg u \land d \land \neg r)\ \mathcal{U}\ (\neg u \land \neg d \land r))) \qquad (4.4.9)$$

The second formula (describing correct behavior as well) ensures that the dd anomaly and du anomalies do not occur. The formula here demands that a variable that is defined shall be referenced at some point after the definition. Any other action, the undefinition or redefinition imply the dd or du anomalies.

$$\phi := \Box(d \rightarrow ((u \wedge \neg d \wedge \neg r)\, \mathcal{U}\, (\neg u \wedge d \wedge \neg r)\, \mathcal{U}\, (\neg u \wedge \neg d \wedge r))) \tag{4.4.10}$$

Note: we have to insert interruption events, for example, to notice dd anomalies.

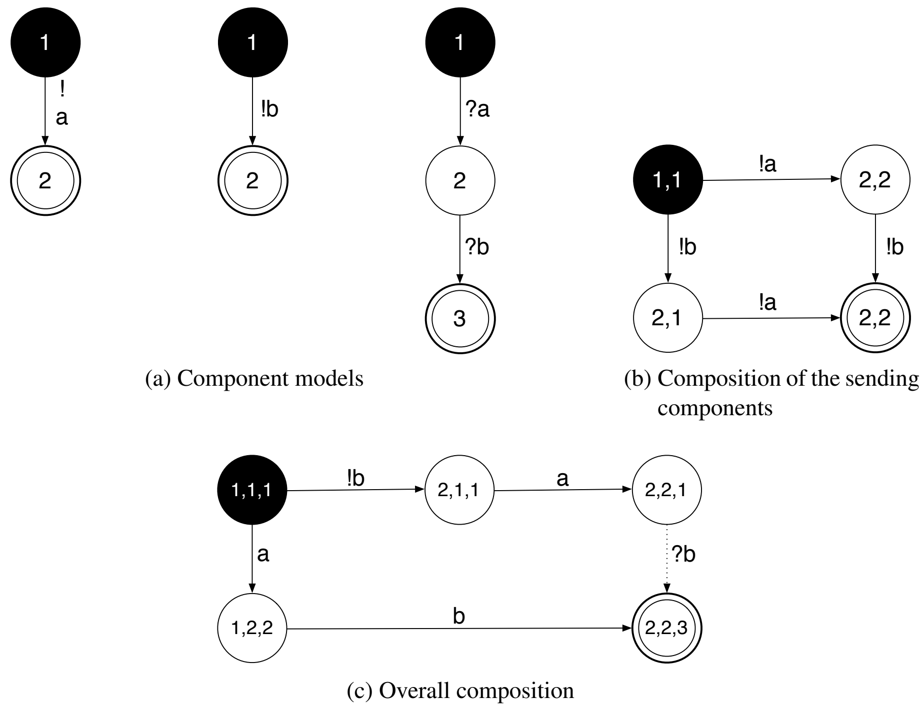
4.4.3.2. Illegal Deadlocks

Description: One characteristic of a test is that unlike a non-terminating reactive system, a test should always end. Two anomalies that usually occur in concurrent contexts are deadlocks and livelocks.

Deadlocks are anomalies where the behavior gets stuck and cannot continue, for example, because a test component is waiting for a message of another test component which will not provide the required message for some reason. In fact, deadlocks in test cases are formally valid in terms of LTSs: test cases must terminate and, therefore, by Definition 2.5, they must have states with no outgoing transitions enabled. Therefore, we need to differentiate between illegal and legal deadlocks and mark these states which legally deadlock.

Example: The classical example for a deadlock is the dining philosophers problem [41] which can in fact also be modeled for communicating message-passing systems. However, for demonstrational purposes a smaller example is sufficient.

Figure 4.13a illustrates three test components. The two components on the left send messages whereas the right-most component receives messages. The composite of the two sending components (Figure 4.13b) can send $!a$ and $!b$ in any order while the receiving component expects $!a$ first. A composition of the overall is represented in Figure 4.13c. Here, we omit all states that are of no interest and unreachable in our scenario. The transitions without the ! or ? signs as prefixes are synchronized transitions between the components, i.e., they represent the behaviors where a message is sent and directly consumed from the queue. Obviously, if the order is $!a!b$ for the sending components, the transition can be synchronized and they reach a state that is marked as a final state (note: to identify valid from invalid end states, we must additionally mark valid end states). On the other hand, if $!b$ is sent first, the behavior continues also without synchronization due to the queued behavior. If we assume a queue size of 1, message $!a$ will arrive after $!b$ without consumption and the $!b$ entry will be lost. Once the behavior reaches state $(2,2,1)$, the behavior deadlocks, since the $!b$ message was dropped—the $?b$ transition cannot take place anymore (hence, the transition is illustrated in a dotted notation). If we assume queue sizes greater than one, the deadlock can occur earlier: If $!b$ is sent first, it cannot be received on the receiving component as it expects $?a$ first. Therefore, the behavior deadlocks right after $!b$ has been sent.



(a) Component models

(b) Composition of the sending components

(c) Overall composition

Figure 4.13.: Example: Deadlock

Analysis: The deadlock analysis is solved by state space exploration of the parallel behavior. States that have no outgoing transitions are deadlock states.

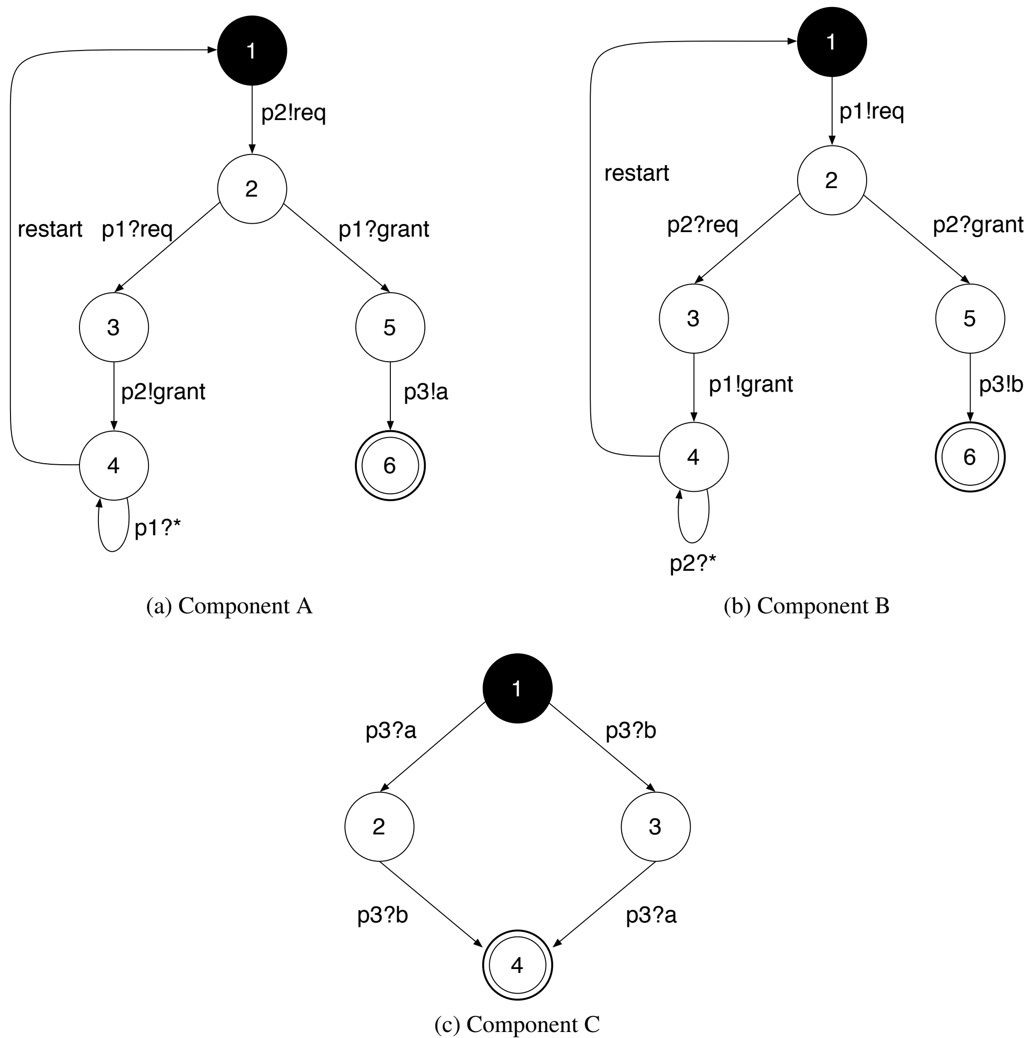
4.4.3.3. Livelocks

Description: Like deadlocks, livelocks also cause processes to get stuck. However, when a livelock occurs, the test components are not waiting for other test components in a single state, but they are continuously working and move forward all the time. However, the work they do in their behavior will never terminate and they are not progressing.

Example: The classical example for a livelock is two people meeting in a narrow corridor. Each one is trying to be polite and steps aside to let the other pass. However, both people end up being polite and move aside again and again at the same time. The situation can only be resolved with one person giving up the politeness.

The same situation can be translated to the context of communicating systems. Imagine a setup of three components *A* (Figure 4.14a), *B* (Figure 4.14b), and *C* (Figure 4.14c). Components *A* and *B* both want to send a message to component *C*: component *A* wants to send message *a* through port *p*3 to *C* and component B wants to send message *b* through port *p*3 to *C*. For component *C*, it does not make any difference whether *A* or *B* send a message

first. However, *A* and *B* must coordinate among each other who goes first, because the messages should not be sent at the same time. We can imagine a situation where the queue length of *C* is limited and we want to avoid an overflow—whichever component sends its message as second has to wait for a few seconds. For that coordination, both components *A* and *B* send requests to each other asking for permission to send. After that, the components either expect a request from the other component or a grant message allowing to send the message to component *C*. Both components prioritize the answer of send requests over accepting grant permissions. Therefore, both components *A* and *B* keep granting each other



(a) Component A

(b) Component B

(c) Component C

Figure 4.14.: Example: Livelock

permission to send messages to C and repeat this process indefinitely. Note: the self-loop in state 4 indicates the consumption of remaining queue items in case the components are composed with queues.

Analysis: The livelock analysis is dealt with by state space exploration. For that kind of liveness analysis, it is necessary to mark certain transitions as progressing transitions. Otherwise, it is impossible for the analysis to figure out when it is stuck in a loop. In the case of the example, such progressing transitions are the transitions $(5, p3!a, 6)$ in component A and $(5, p3!a, 6)$ in component B.

4.4.3.4. Illegal Double Calls

Description: Specific kinds of subsequently following events of the same kind may indicate misusage in specific contexts. In principle, this a generalized form of the dd anomaly which can also play a role for events that do not necessarily represent data manipulations. Certain other events have the capability to reset the situation.

Example: It is a good idea to have a look at a concrete testing language to find a reasonable example for this anomaly. In TTCN-3, for instance, among events that should not directly occur subsequently are *start* statements if there is no *done* statement in between. Another example is timer handling on alive components. If a TTCN-3 alive component is started multiple times subsequently, there could be the requirement that possible component timers are reset in between in order to avoid unexpected timeouts as component timers keep running on alive components even when its behavior terminates.

Figure 4.15 illustrates such a scenario. Two boolean variables are used to model this problem: the e variable models the event that should not occur multiple times subsequently. The r variable represents the event that resets the scenario, i.e., an e event may take place again after the r event happened. In the figure, a component is defined, created, and then started. Depending whether some condition holds, either the left or right branch is chosen. In the left branch, the same component is started again and afterwards, the done statement is called. If the *someCondition* does not hold, the done statement is called and the component is started again. The left branch represents the kind of illegal double call that we want to avoid, whereas the right branch represents a valid sequence regarding the illegal double call property.

Analysis: The LTL formula (describing correct behavior) for the analysis essentially describes that whenever we find a state where an event e takes place, for a following state event r must occur or e must remain true.

$$\phi := \Box\,(e \;\rightarrow\; \Diamond\, r \;\vee\; \Box\, e) \tag{4.4.11}$$

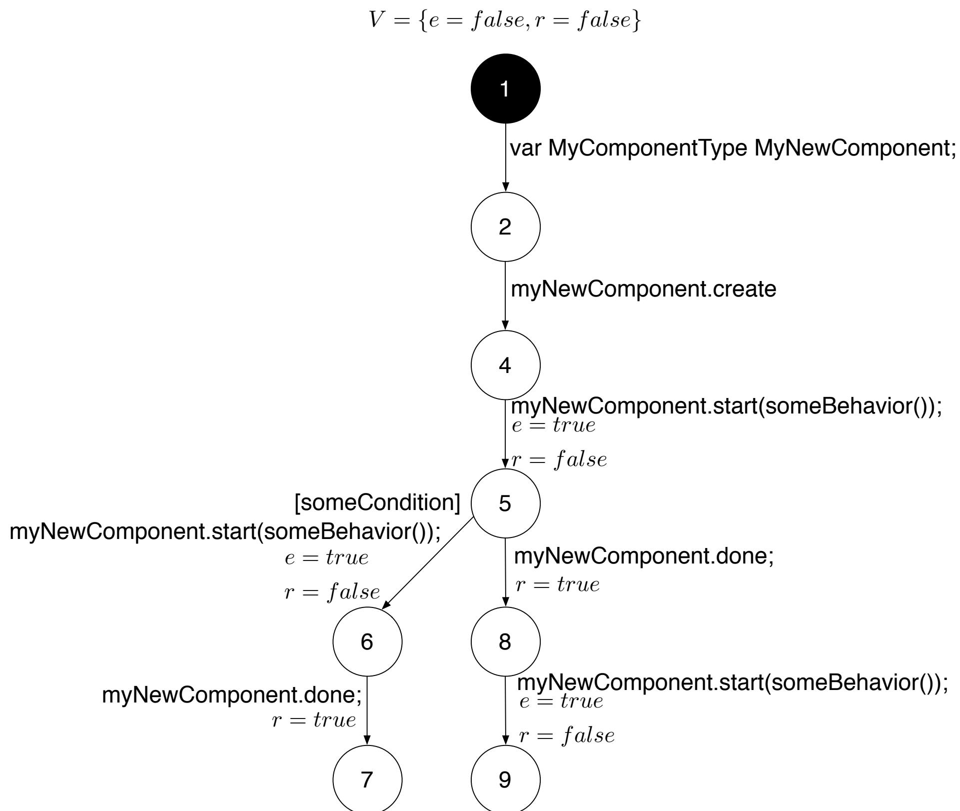


Figure 4.15.: Example: Illegal Double Calls

Just as in the dd anomaly, every $e = true$ event must be directly preceded by an interruption event that sets e to false. Otherwise, we are not able to determine if another e event takes place. The interruption events are not illustrated in the example figure.

4.4.3.5. Event Pair Asymmetry

Description: When a certain event a occurs, it is required that always an event b follows eventually.

Example: For a motivation and an example for this generic anomaly, please refer to the *Idle PTC* anomaly in Section 4.4.2.

Analysis: The proposed LTL formula (describing correct behavior) analyzes that if a certain event a occurs, then b must always occur eventually. The formula corresponds to the global response specification pattern [45].

$$\phi := \Box\,(a \;\rightarrow\; \Diamond\, b) \tag{4.4.12}$$

4.4.4. Limitations

The test case reconstruction and analysis method presented here is practically applicable. The complexity of the model reconstruction method is of linear nature and, therefore, not an issue. Constraints regarding computational limits can only be found in the model checking technology used. Whether this presents a problem depends on the behavior that is analyzed.

The degree of abstraction of the reverse engineered model plays a significant role for the results that can be provided by an application of the presented methods. As more abstraction is introduced, the more we have to deal with false positives, i.e., detected anomalies that are no anomalies in fact. Luckily, when black-box testing communicating systems, the role of internal data of tests seems to be not as big as for general-purpose software. We expect that there is a number of false positives. However, it is likely that this number is not too huge and easy to check manually. Additional tool support to display the faults within the code itself can make the review of the analysis results easier and quicker.

The choice to model events using only boolean variables surprisingly works for a lot of different dynamic anomalies that may occur—despite its minimalistic approach. However, not all problems can be covered easily that way. For example, if we want to analyze TTCN-3 connection violations prior to any execution, it would certainly be possible to model each situation with boolean variables, but the notation would become very complex. A model extension to handle arrays would allow such scenarios to be defined in a more compact way. Chapter 5 describes a observations made when applying an prototype implementation of this approach to standardized test suites of industrial size.

4.5. Response Consistency

The anomalies in the previous section are related to the behavior of single test cases. Test suites are composed of multiple test cases and it is thus interesting to take a look at the big picture as well. The analysis of test suites can have many purposes. One popular analysis target is the extraction of information from test suites in order to select a subset of an existing test suite (test selection). The intention for such an analysis might be the focus on specific feature sets that might have been the subject of change or the reduction of test execution time when the execution of the complete test suite is not feasible with respect to the time that it would require. Such analyses often try to optimize a coverage criterion while keeping the number of test cases minimal.

Another approach to test suite analysis is to look for similarities in different test cases in an attempt to find behaviors that contradict each other or that may indicate a contradiction. Response consistency violations are situations that arise when multiple test cases in a test suite have the same stimuli sequences, but expect different responses after the stimuli have been sent. Using this response consistency criterion, we identify test cases that are contained in each other or that expect completely different responses despite their equivalent stimuli sequences. We use the LTS model to describe this criterion and discuss the

examples. Note that in this analysis, we are only interested in consistency issues regarding the communicating behavior of the test specifications. For that purpose, we make use of a test-specific behavioral pattern that tests for communicating systems typically follow. This repeating pattern is that stimuli are sent and certain responses are expected. Adding verdicts to these kinds of analysis leads to different kinds of analysis (see Section 6.2).

4.5.1. Introductory Example

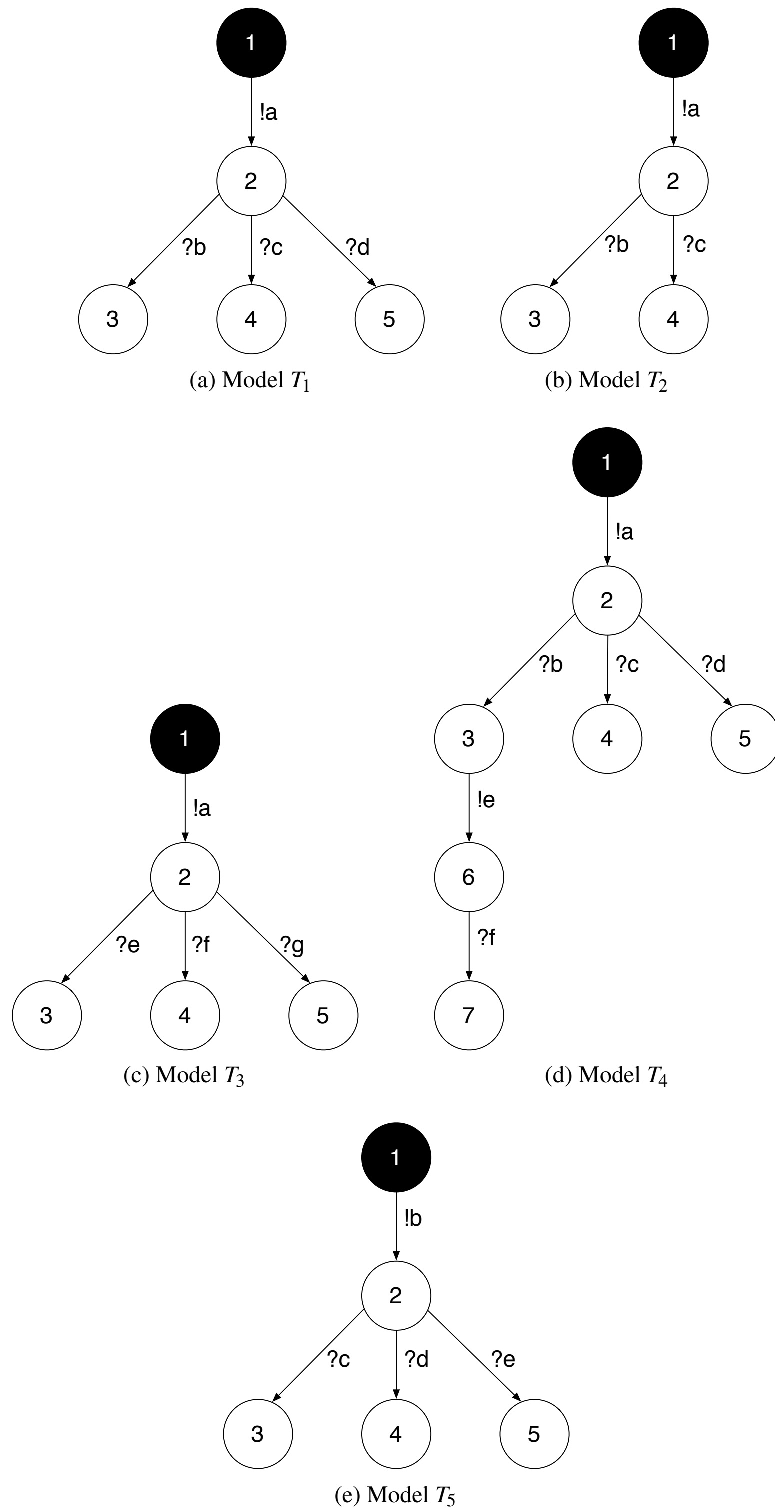
The intuition of a response consistency violation is the following: two communicating models T_1 and T_2 representing the communicating behavior of abstract test cases have some common action sequence σ such that $s_0 \stackrel{\sigma}{\Rightarrow} s$ can be found in T_1 and $s_0 \stackrel{\sigma}{\Rightarrow} s'$ can be found in T_2. When s and s' respectively are states that expect only responses, they should expect the same responses—after all, the preceding action sequence between T_1 and T_2 matched and thus the responses should be the same.

Take for instance the models illustrated in Figure 4.16a and 4.16b. Both models start with a stimulus $!a$ and thus have the same observable prefix. However, in T_1, messages $?b, ?c$, or $?d$ are expected, while T_2 only expects messages $?b$ or $?c$. T_2 expects a subset of the messages that T_1 expects and is essentially contained in T_1. In this situation, it is unclear why T_2 does not handle a possible incoming message d. It may simply be a consistency violation due to a human mistake. We assume that our test cases are written by hand and not generated automatically. On the other hand, assuming that we always begin in the same start state, models T_1 and T_3 (Figure 4.16c) have again the same stimulus $!a$. However, even being in the same state, both test cases expect entirely different responses. Finally, models T_1 and T_4 (Figure 4.16d) have the same stimulus $!a$ and expect the same messages $?b, ?c$, and $?d$, but T_4 continues with additional behavior once $?b$ is received.

In general, such models with the same preceding stimulus sequence may differ in a receiving state in three distinguishable ways:

- The test cases expect the same follow-up responses (models T_1 and T_4).
- One test case handles a subset of the expected messages of another test case (models T_1 and T_2).
- One test case expects completely different messages than another test case. The receive sets are disjoint (models T_1 and T_3).

The first case presents a response consistent scenario. The second and third case present a response consistency violation. The general underlying assumption, in this local scenario, is that a test case is initiated by stimuli, responses follow in answer to the stimuli, and then again possibly a new stimuli–response sequence is initiated or the test case ends. In other words, we assume that we deal with test cases that have a repeating stimuli–response pattern. Within these patterns, we want to find contradictions in the responses by comparing test case pairs. Of course, the scenarios presented in Figure 4.16 are simple cases. In



(a) Model T_1 (b) Model T_2

(c) Model T_3 (d) Model T_4

(e) Model T_5

Figure 4.16.: Example: Response Inconsistency

practice, we deal with test cases that may also have varying response orders or concurrent behavior.

4.5.2. Response Consistency Definition

For the purpose of identifying response consistency or its violation by a test case pair, we define a binary relation that describes response consistency of a test case pair. Informally, two distinct cases are distinguished:

- The test cases are observationally equivalent in their communicating behavior. This situation may happen, for example, when different aspects of the message exchange are checked in two separate test cases.
- The test cases have coinciding stimuli sequence prefixes and the stimuli responses within this prefix are consistent (models T_1 and T_4).

If two test cases are observationally equivalent, there are no contradictions in the responses. The traditional relations to describe such kinds of observational equivalence are trace equivalence or bisimulation [104] (Section 2.2.4). Both relations describe systems whose observable moves cannot be distinguished from each other. The bisimulation relation, is also able to distinguish non-deterministic systems, which the trace equivalence relation cannot. A third undefined case for response consistency is when two test cases have entirely different stimuli sequences (models T_1 and T_5 in Figure (Figure 4.16e)). In this case, the test cases are strictly no candidates for the analysis.

For establishing a notion of response consistency, we relate test cases in a symmetric fashion, i.e., if a test case A is response consistent to a test case B, then B is also response consistent to A. Thus, preorders such as trace preorder (Definition 2.13) are insufficient. Arguably, for test cases it may be enough to establish trace equivalence due to the assumption that test cases are supposed to be deterministic. Bisimulation on the other hand primarily overcomes the limitations of trace equivalence when non-determinism must be taken into consideration. In that sense, bisimulation is a stronger, but also a more exact criterion.

We assume that response contradictions only occur when the stimuli sequences coincide—assuming that the SUT behaves deterministically in case it is initialized to the same start state and the same stimuli sequence is consumed. By stimuli sequences, we formally mean rewritten traces where only these actions are concatenated that are outputs from the test case. For this purpose, we define the *stimuliseq* operator that essentially slices all actions from a trace except for the stimuli.

$$stimuliseq(a_1 \cdot a_2 \cdot \ldots \cdot a_n) := \forall a_i, a_j \in A_O, i < j : a_i \cdot a_j$$

After defining the *stimuliseq* operator, we can provide a definition for response consistency.

Definition 4.5 (Response Consistency (reco)) *Let T_1 and T_2 be two LTSs. Then T_1 and T_2 are response consistent, or T_1 **reco** T_2, iff $\exists s_0 \overset{\sigma}{\Rightarrow} s, \sigma \in traces(T_1)$ and $\exists t_0 \overset{\varsigma}{\Rightarrow} t, \varsigma \in traces(T_2) : stimuliseq(\sigma) = stimuliseq(\varsigma)$ and the following conditions hold:*

- *for all $s \overset{a}{\Rightarrow} s'$ with $a \in A_I^{T_1}$, there exists a $t \overset{a}{\Rightarrow} t'$ with $a \in A_I^{T_2}$ and*
- *vice versa: for all $t \overset{a'}{\Rightarrow} t'$ with $a' \in A_I^{T_2}$, there exists an $s \overset{a}{\Rightarrow} s'$ with $a \in A_I^{T_1}$.*

If there are no matching stimuli sequences, the response consistency is undefined.

In comparison to an observational equivalence relation such as the weak bisimulation relation, the **reco** relation is weak in its condition. It essentially demands that the stimuli sequences are entirely different or that for all states in a test case T_1 reachable by a common stimuli sequence and with an outgoing response transition, there must be a corresponding response transition in T_2 to a state that is reachable by the same stimuli sequence and vice versa. In addition, the matching stimuli sequence condition implies that test cases can be response consistent even if they drift apart in their stimuli at some point. The consistency criterion is only concerned with those states that are reachable by coinciding stimuli sequences. The weak bisimulation or trace equivalence relations are essentially borderline cases of the **reco** condition. Two response consistent test cases are weakly bisimilar when the stimuli sequences for all traces symmetrically match and the response orders are the same [155].

If we again take a look at the examples depicted in Figure 4.16, we notice that T_1 **reco** T_2 is false. The test cases have an equivalent stimulus prefix $!a$ and, therefore, the **reco** condition must hold. But it fails. There are no traces with the same stimulus sequence where the responses between the stimuli match completely. The traces where the stimuli sequence matches reach state 2,3,4, or 5 in T_1. However, a corresponding transition in T_2 for $?d$ is missing. For T_2, however, there are corresponding response transitions in state T_1. As the **reco** condition is symmetric, the condition does not hold and T_1 and T_2 thus violate the response consistency criterion. Similarly, T_1 **reco** T_3 is false (the **reco** condition fails), T_1 **reco** T_4 is true (the **reco** condition holds) and T_1 **reco** T_5 is undefined.

The notion of the **reco** relation disregarding all transitions in between the stimuli (i.e., responses and internal transitions) is based on the assumption that the SUT behaves deterministically to the stimuli and that, always starting in the same start state, the same stimuli sequences steer the SUT into the same SUT state. Two test cases might theoretically have the same stimulus, the same responses to this first stimulus and then again the same stimulus, but attached as transitions to states reached by different responses. While this situation may seem awkward, the **reco** relation in fact considers these two test cases to be response consistent if the responses after the second coinciding stimulus are again consistent. The reason is that we will not be confronted with a follow-up comparison of responses after a second stimulus when the responses to the first stimulus are different if the SUT responds deterministically. For non-deterministic SUTs, a stronger condition than **reco** is needed

that somehow respects the fact that the responses between the stimuli must match as well in some way. Such a relation for non-deterministic SUTs can be found in [155].

4.5.3. Response Consistency Detection Algorithm

The following algorithm detects whether two test cases T_1 and T_2 are response consistent and the direction of possible response consistency violations.

Algorithm 4.1 Given are two models T_1 and T_2. Let aseq $:= a_i \cdot a_{i+1} \cdot \ldots \cdot a_j$ with $a_i, \ldots, a_j \in T_{i_{A_O}}$ be sequence of output actions in T_i and ASEQ_{T_i} be the set containing all aseq in T_i. Furthermore, let $\mathrm{SSEQ}_{T_i} \subseteq \mathrm{ASEQ}_{T_i} \times S_{T_i}$ be a relation that relates sequences of actions in T_i to states S_{T_i} in T_i. Perform the following steps to detect whether T_1 violates response consistency regarding T_2 or the other way round:

- For T_1 and T_2, perform a depth-first search (outer search) that cancels on visited nodes. During the search, track the stimuli so that all sequences aseq_{T_i} are identified and that a mapping to the respective target state s' after transitions $(s, a, s'), a \in T_{i_{A_O}}$ are stored in SSEQ_{T_i}.
- If $\nexists v_i, v_j$ with $v_i = v_j$ and $(v_i, s) \in \mathrm{SSEQ}_{T_1}, s \in T_{1_S}$ and $(v_j, t) \in \mathrm{SSEQ}_{T_2}, t \in T_{2_S}$, then the consistency between T_1 and T_2 is undefined. Stop the algorithm.
- Otherwise, for each v_i with $(v_i, s) \in \mathrm{SSEQ}_{1_S}$ and v_j with $(v_j, t) \in \mathrm{SSEQ}_{2_S}$ and $v_i = v_j$, perform the following steps:
  - For T_1 and T_2, perform a depth-first search (inner search) starting in state $s \in T_1$ and $t \in T_j$, with $(v_i, s) \in \mathrm{SSEQ}_{1_S}$ and $(v_j, t) \in \mathrm{SSEQ}_{2_S}$, that cancels the traversal when the transitions to be traversed are not internal or input transitions.
  - Collect each input action that takes place in a transition visited during the search in sets $\mathrm{RESPONSES}_{v_i}$ and $\mathrm{RESPONSES}_{v_j}$.
  - If $\mathrm{RESPONSES}_{v_i} = \mathrm{RESPONSES}_{v_j}$, the responses for the stimulus sequence are consistent. Continue with the next pair v_i, v_j with $v_i = v_j$.
  - If $\mathrm{RESPONSES}_{v_i} \subset \mathrm{RESPONSES}_{v_j}$ or $\mathrm{RESPONSES}_{v_j} \supset \mathrm{RESPONSES}_{v_i}$, then the responses are inconsistent. Stop the algorithm.
- If the algorithm terminates and has not been stopped early, then T_1 and T_2 are response consistent.

The inner depth-first searches do not necessarily have to be performed separately, but can be incorporated into the outer search as well. Therefore, the overall complexity of the response inconsistency detection of two test cases has the worst case performance of two times a depth-first search, i.e., $O(|S| + |\lambda|)$. However, in practice, we will probably compare all test cases of a test suite among each other.

4.5.4. Scenario: Sequential Models with Different Response Orders

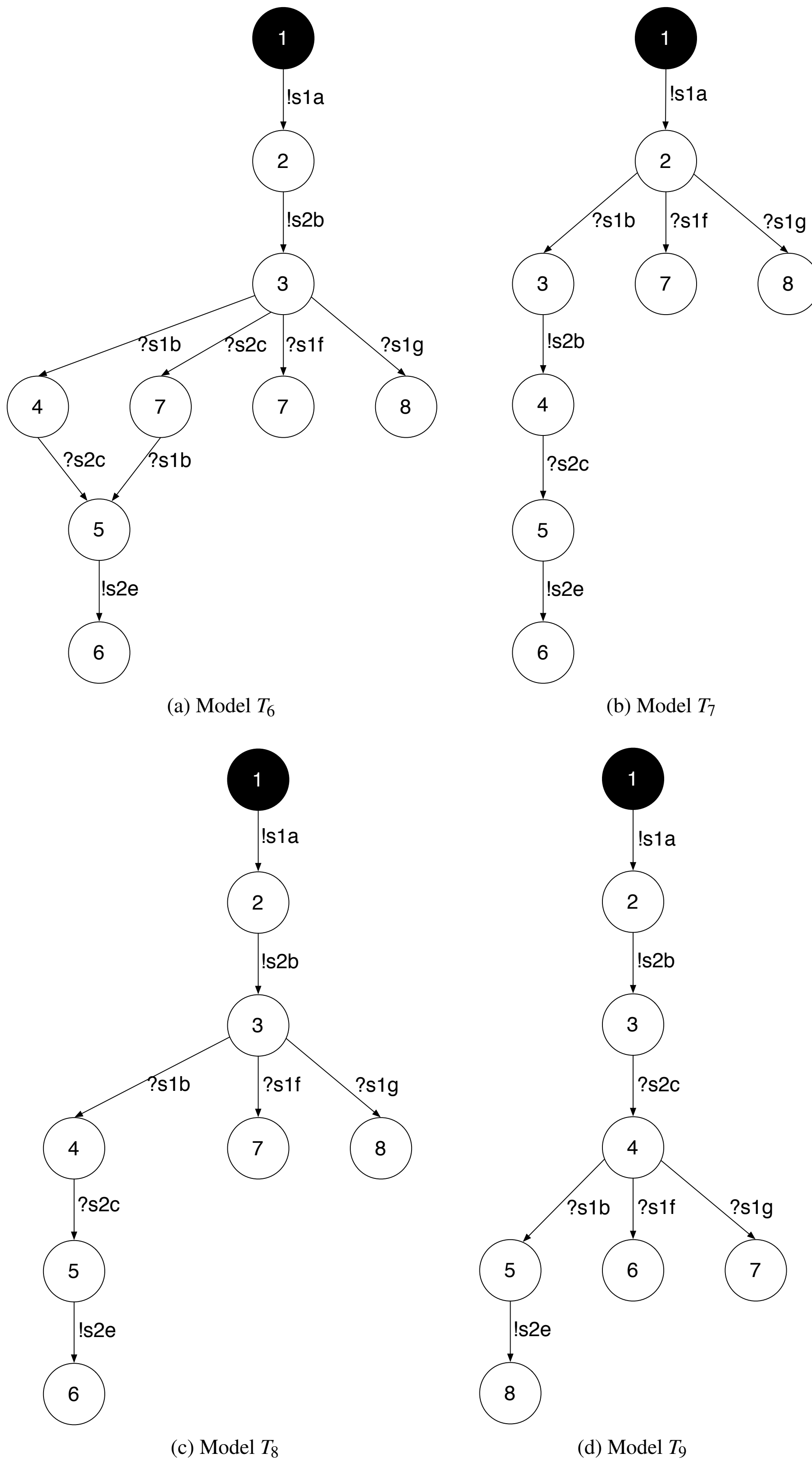
The provided examples so far always assumed that the response events of the compared models are ordered in the same way. Similarly, we have only discussed local models without concurrent behavior. In general, we need to distinguish the following cases when we compare models for response consistency violations:

- We compare a local model to another local model where both have the same response orders.
- We compare a local model to another local model where both have different response orders.
- We compare a concurrent model to a local model.
- We compare two concurrent models.

The differentiation already indicates that there is a high degree of variety how test cases and their models can be built even though they essentially provide the same behavior. For example, models with concurrent behavior can produce the same behavior as a local model. Similarly, deterministic local models can systematically regard different message reception orders. We explicitly want to state that we do not find every kind of test design mentioned above and in the following reasonable for real-world testing. However, they may occur and are practically possible.

We continue the discussion by relating models T_6 and T_7 (Figures 4.17a and 4.17b) regarding inconsistent responses. Both are local models which have the same stimuli sequences ($!s1a$, $!s1a \cdot !s2b$, and $!s1a \cdot !s2b \cdot !s2e$), but with a different response order in between. T_6 sends the first two stimuli $!s1a \cdot !s2b$ subsequently and then expects the responses $?s1c$, $?s2c$, $?s1f$, and $?s1g$. If either $?s1b$ or $?s2c$ is received, the respective other message is expected right afterwards (i.e., the order of $?s1b$ and $?s2c$ does not matter) and a third $!s2e$ stimulus is sent. In T_7, first the $!s1a$ stimulus is sent and $?s1b$, $?s1f$, and $?s1g$ are expected. Only if $?s1b$ is received, a second $!s2b$ is sent and the response $?s2c$ is expected. If $?s2c$ is received, the third stimulus $!s2e$ is sent.

A more intuitive study of the models T_6 and T_7 might lead to the conclusion that they essentially do the same, that the stimuli $!s1a$ and $!s2b$ are in fact independent, and that causal relationships only exist between the respective stimuli and responses. However, such intuitive assumptions might be misleading. For example, the transitions $?s1f$ and $?s1g$ cannot be consumed in T_6 if $?s2c$ is received first. We have to be careful and assume that T_6 and T_7 model in fact different things for a reason. Hence, the **reco** relation fails for T_6 and T_7: both models have the same stimuli sequence $!s1a$, but reaching state 2, there is a mismatch as T_6 has no transitions for the responses that take place in T_7. Applying the response consistency detection algorithm yields the comparisons between $v_i = !s1a, (v_i, 2) \in \text{SSEQ}_{T_6}$ and $v_j = !s1a, (v_j, 2) \in \text{SSEQ}_{T_7}$ where $\text{RESPONSES}_{v_i} = \{\}$ and $\text{RESPONSES}_{v_j} = \{?s1b, ?s1f, ?s1g\}$. As $\text{RESPONSES}_{v_i} \neq \text{RESPONSES}_{v_j}$, the models violate response consistency and further algorithm iterations are not necessary.



(a) Model T_6

(b) Model T_7

(c) Model T_8

(d) Model T_9

Figure 4.17.: Local Test Cases with Different Response Orders

The situation is different for models T_8 and T_9 (Figures 4.19 and 4.17d). They have again the same stimulus sequences $!s1a$, $!s1a{\cdot}!s2b$, and $!s1a{\cdot}!s2b{\cdot}!s2e$. After the $!s1a$ stimulus, there are no responses in both models, but a second stimulus $!s2b$ is sent immediately, i.e., the responses match after the first stimulus. After the second stimulus, T_8 expects $?s1b$, $?s1f$, and $?s1g$. If $?s1b$ is received, $?s2c$ is expected next. In T_9, $?s2c$ is expected first and only when it arrives are $?s1b$, $?s1f$, and $?s1g$ expected. T_8 and T_9 are in fact **reco** as the responses after $!s1a$ are in both cases none and the responses after the second stimulus $!s2b$ can be reached respectively. Finally, both test cases again do not expect any responses after $!s1a{\cdot}!s2b{\cdot}!s2e$. Applying the response consistency detection algorithm yields the comparisons:

- $v_i = !s1a, (v_i, 2) \in \text{SSEQ}_{T_8}$ and $v_j = !s1a, (v_j, 2) \in \text{SSEQ}_{T_9}$ with $\text{RESPONSES}_{v_i} = \{\}$ and $\text{RESPONSES}_{v_j} = \{\}$, i.e., $\text{RESPONSES}_{v_i} = \text{RESPONSES}_{v_j}$.
- $v_i = !s1a{\cdot}!s2b, (v_i, 3) \in \text{SSEQ}_{T_8}$ and $v_j = !s1a{\cdot}!s2b, (v_j, 3) \in \text{SSEQ}_{T_9}$ with $\text{RESPONSES}_{v_i} = \{?s1b, ?s1f, ?s1g, ?s2c\}$ and $\text{RESPONSES}_{v_j} = \{?s2c, ?s1b, ?s1f, ?s1g\}$, i.e., $\text{RESPONSES}_{v_i} = \text{RESPONSES}_{v_j}$.
- $v_i = !s1a{\cdot}!s2b{\cdot}!s2e, (v_i, 6) \in \text{SSEQ}_{T_8}$ and $v_j = !s1a{\cdot}!s2b{\cdot}!s2e, (v_j, 8) \in \text{SSEQ}_{T_9}$ with $\text{RESPONSES}_{v_i} = \{\}$ and $\text{RESPONSES}_{v_j} = \{\}$, i.e., $\text{RESPONSES}_{v_i} = \text{RESPONSES}_{v_j}$.

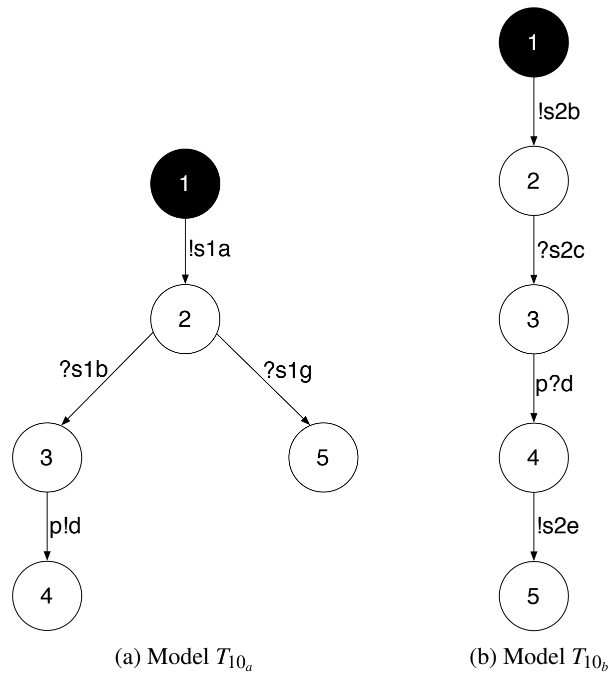
The latter example illustrates nicely how **reco** disregards structural properties among the response receival orders.

4.5.5. Scenario: Concurrent Models

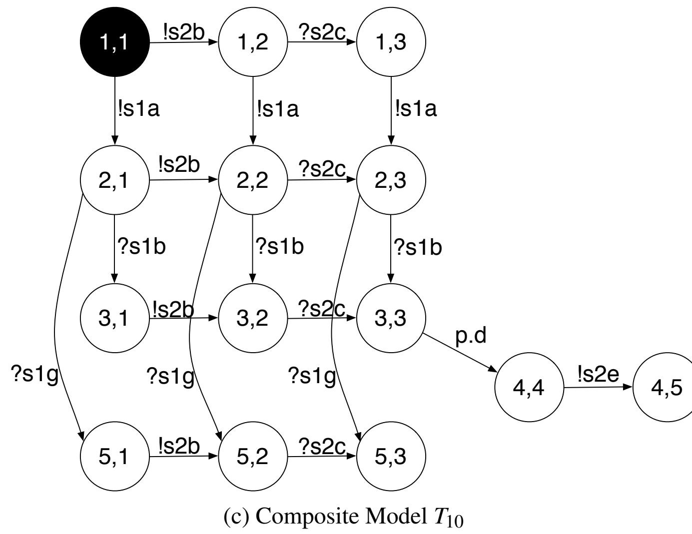
Test cases are often written with the help of test components that are executed concurrently. Here, the determination of response consistency violations is not as intuitive since the behavior is defined by composite models where non-determinisms can occur easily due to their interleaving structure.

Figure 4.18 illustrates an example for such a concurrent model. Figures 4.18a and 4.18b show the local behavior of two test component models. T_{10_b} can only send the message $!s2e$ when its behavior is synchronized with T_{10_a} through $p?d$ and $p!d$, i.e., the message must only be sent if $?s1b$ was received in T_{10_a}. The composite model $T_{10} = T_{10_a} \| T_{10_b}$ according to Definition 2.8 is presented in Figure 4.18c. To reduce the size of the figure, we omitted all states and transitions that are not reachable from the start state. The order of the events depends on which events are independent from each other and which events are dependent. For example, $!s1a$ must take place before $?s1b$ or $?s1g$, but $!s2b$ may take place any time in between. Furthermore, in the composite model, we have non-determinisms between inputs and outputs (e.g., in state $(2,1)$) and also between multiple outputs (e.g., in state $(1,1)$).

Based on the observations discussed in Section 4.5.4, we can test cases with local or local with different response orders designs against test cases with a concurrent design as



(a) Model T_{10_a} (b) Model T_{10_b}



(c) Composite Model T_{10}

Figure 4.18.: Model T_{10}

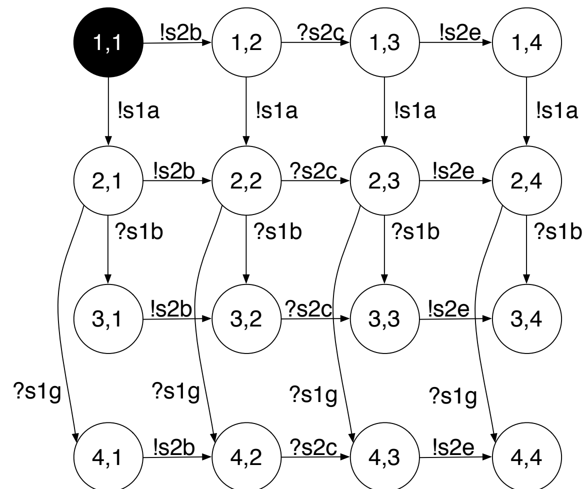


Figure 4.19.: Model T_{11}: Composite Model T_{10} Without Synchronization

well. There is no limitation in the applicability of the **reco** relation. We demonstrate this by example.

We compare test cases T_6 and T_{10}. Both contain the stimuli sequences $!s1a$, $!s1a{\cdot}!s2b$, and $!s1a{\cdot}!s2b{\cdot}!s2e$. T_{10} contains the additional stimuli sequences $!s2b$, $!s2b{\cdot}!s1a$, $!s2b{\cdot}!s1a{\cdot}!s2e$ that are not part of T_6 and hence are left disregarded as the **reco** relation only regards traces that exhibit equal stimuli sequences. In every observable trace having the stimulus prefix $!s1a$ in T_6, there are no corresponding responses. After the $!s1a{\cdot}!s2b$ stimuli sequence in T_6, there are states where every response can be consumed, i.e., $?s1b, ?s1f, ?s1g$, and $?s2c$. Finally, after the $?s1a{\cdot}!s2b{\cdot}!s2e$ stimuli sequence, there are again no states in which responses are consumable. In T_{10}, the consumable responses are quite different. After $!s1a$, there are states in which $?s1b$ and $?s1g$ are expected. The stimuli sequence $!s1a{\cdot}!s2b$ leads to states that can consume $?s1b, ?s1g$, and $?s2c$. Finally, after $!s1a{\cdot}!s2b{\cdot}!s2e$, there may be no more responses in T_{10}. When evaluating the possible responses after traces with the same stimuli, we notice that the $!s1a$ sequence delivers a response mismatch where the response sets contradict each other, i.e., the response set for T_6 is empty while it is non-empty for T_{10}. Therefore, T_6 is not **reco** with T_{10}.

The comparison between T_6 and T_{10} indicates that the comparison between local and concurrent test cases often exhibit inconsistencies and they are not necessarily considered a possible anomaly from the point of view of the test developer. Testers who write test cases with concurrent test behavior accept or even disregard the fact that the traces in subsequent executions of the same test may vary. For T_6, a possible interpretation is that the stimuli are independent from each other and the responses on $s1$ and $s2$ are independent from each other as well. However, the test case design is to send both stimuli $!s1a$ and $!s2b$ before expecting the responses for both stimuli rather than send-

ing $!s1a$, then handling the responses for $!s1a$, and then sending $!s2b$ before handling the responses for $!s2b$ (this is what test case T_7 in Figure 4.17b illustrates). Applying the response consistency detection algorithm yields at the very beginning the comparison $v_i = !s1a, (v_i, 2) \in \text{SSEQ}_{T_6}$ and $v_j = !s1a, (v_j, (2,1)) \in \text{SSEQ}_{T_{10}}$ with $\text{RESPONSES}_{v_i} = \{\}$ and $\text{RESPONSES}_{v_j} = \{?s1g, ?s1b\}$, i.e., $\text{RESPONSES}_{v_i} \neq \text{RESPONSES}_{v_j}$. Further algorithm iterations are not necessary.

We conclude the discussion with a comparison between two concurrent test cases. T_{11} (Figure 4.19) is essentially the composition of test cases T_{10_a} and T_{10_b} without the $p!d$ and $p?d$ transitions that synchronize the behavior. The purpose of this synchronization is to wait with the $!s2e$ transition in T_{10_b} until $?s1b$ or $p!d$ respectively took place in T_{10_a}. Removing this synchronization essentially means that $!s2e$ can take place any time after $?s2c$ was received in T_{10_b}. As a result, the $?s1b$ and $?s1g$ responses may still take place after the $!s2e$ transition took place in T_{11} and thus the response set of the $!s1a{\cdot}!s2b{\cdot}!s2e$ stimuli sequence is not empty for T_{11}, but there exist states which consume the responses $?s1b$ and $?s1g$ after the stimuli sequence took place. In T_{10}, such states do not exist after the same stimuli sequence and thus, the **reco** condition for T_{10} and T_{11} does not hold. Applying the response consistency detection algorithm yields the comparisons:

- $v_i = !s1a, (v_i, (2,1)) \in \text{SSEQ}_{T_{10}}$ and $v_j = !s1a, (v_j, (2,1)) \in \text{SSEQ}_{T_{11}}$ with $\text{RESPONSES}_{v_i} = \{?s1g, ?s1b\}$ and $\text{RESPONSES}_{v_j} = \{?s1g, ?s1b\}$, i.e., $\text{RESPONSES}_{v_i} = \text{RESPONSES}_{v_j}$.
- $v_i = !s1a{\cdot}!s2b, (v_i, (2,2)) \in \text{SSEQ}_{T_{10}}$ and $v_j = !s1a{\cdot}!s2b, (v_j, (2,2)) \in \text{SSEQ}_{T_{11}}$ with $\text{RESPONSES}_{v_i} = \{?s1g, ?s1b, ?s2c\}$ and $\text{RESPONSES}_{v_j} = \{?s1g, ?s1b, ?s2c\}$, i.e., $\text{RESPONSES}_{v_i} = \text{RESPONSES}_{v_j}$.
- $v_i = !s1a{\cdot}!s2b{\cdot}!s2e, (v_i, (4,5)) \in \text{SSEQ}_{T_{10}}$ and $v_j = !s1a{\cdot}!s2b{\cdot}!s2e, (v_j, (2,4)) \in \text{SSEQ}_{T_{11}}$ with $\text{RESPONSES}_{v_i} = \{\}$ and $\text{RESPONSES}_{v_j} = \{?s1g, ?s1b\}$, i.e., $\text{RESPONSES}_{v_i} \neq \text{RESPONSES}_{v_j}$.

Thus, the last comparison indicated an inconsistency and further algorithm iterations are not necessary (checking the stimuli permutations). The example with T_{10} and T_{11} illustrates that even small changes in concurrent behavior can have a huge impact regarding the overall behavior. In this case **reco** conforms to the intuition that the model behavior is in fact different. T_{10} implies some causal relation between the $!s2e$ message and the behavior that precedes it on both test components, whereas in T_{11} this is not the case and the causal relationships are only defined by the respective local test components.

4.5.6. Limitations

The detection of response consistency violations is practically feasible and can be applied to the models as we extract them in our case study (Chapter 5). However, there are two things to keep in mind when applying it. First, the **reco** relation assumes that the SUT behaves deterministically regarding the stimuli. For non-deterministic SUTs, the responses do not

necessarily depend on the stimulus sequence alone, but the reactions may also depend on other factors. For non-deterministic SUTs, a different approach towards the detection of response consistency violations has to be taken.

The comparison of concurrent models may not always exhibit the behavior that the test developer anticipates. Concurrent behavior often introduces additional responses and response orders that are not available in local models. The local models may imply causal relationships between behaviors that cannot be ignored in an analysis and yet may not have any meaning and simply exist due to the sequential nature of the test case. Therefore, the detection of response consistency violations may correctly classify test cases as inconsistent whereas their test developer would consider them to be actually consistent, ignoring the fact that these different response orders are possible in the concurrent behavior as well.

4.6. Related Work

In general, there is no known work on the reverse engineering of behavioral models of test case behavior. This aspect of this thesis is unique so far. However, there is a great amount of work regarding reverse engineering of behavioral and structural models (design recovery) and the reverse engineering of models for the purpose of model checking, of which we present a selection in the following.

Briand et al. [25] present an approach for the reverse engineering of UML sequence diagrams from execution traces using formal transformation rules. The execution traces are gathered from the execution of system level test cases against the system whose model is the target of the reverse engineering. From these executions, information is logged by means of instrumented system code. The purpose of their design recovery is to improve the understandability of the reverse engineered artifacts.

A similar approach to reverse engineer structural diagrams is taken by Flanagan et al. [62]. They reverse engineer structural object models also by instrumenting the Java classes under analysis and then executing unit tests against the system under analysis. Their particular focus is the extraction of properties from analyzing heaps of object allocations.

Hamou-Lhadj et al. [72] recover use-case maps from execution traces. Use-case maps are high-level behavioral models which focus on causal sequences of responsibilities and abstract from message exchanges in comparison to UML sequence diagrams. They suggested the removal of utility components to reduce the verbosity of the reverse engineering models, thus allowing an easier understanding. Walkinshaw et al. [151] use symbolic execution to reverse engineer state transitions and how states are related from Java source code. Their proof of concept implementation is an extension of the Java Pathfinder [101].

More related to our work are reverse engineering methods that have the verification of properties of software as target (software model checking). Holzmann and Smith [80, 81] describe a model extraction tool from a modified C parser that extracts the control-flow

graph of the C code and performs data-dependency analysis in addition to generate the verification model.

Corbett [37] describes a method to extract finite state machines statically from program source code. The reverse engineering models are specialized directly for the properties that should be verified later, i.e., their reverse engineering abstraction is tailored towards the verification properties and they slice behavior that is unnecessary. Havelund presents a translation from Java to Promela [73]. His work does not involve an intermediate model, but is a direct translation into the input language of the Spin model checker [79]. Later versions of the Java Pathfinder perform model checking directly on models derived from the Java bytecode.

Ulrich, Petrenko, Boroday, and Hallal have published several papers and articles on the reverse engineering of models from traces of distributed systems [69, 70, 142]. The model reconstruction is achieved by inferring causal relationships between events in the traces. The reverse engineered models are then verified for certain properties using a model checker.

Work on the analysis of test suites, as opposed to test cases, is rare as well. Test selection can be regarded as a method that involves the analysis of test cases in a test suite, as there must be some specific similarity measure between them that allows a decision whether to omit a test case from an test execution or whether a test case should be included in a test run. The available work on test selection as a topic is enormous. For example, Cartaxo et al. [29] describe a similarity function based on the observed number of identical transitions. Similarly, Alilovic-Curgus and Vuong [5] have proposed a distance metric that penalizes mismatching symbols in execution sequences. The overall number of proposals to measure test case similarity for test case selection is too high to list individually. Utting and Legeard [144] provide a general overview of the criteria involved and the corresponding literature. To our knowledge, there is no work that bases a similarity measure on equivalent stimuli sequences though. Equivalent stimuli sequences, however, are primarily interesting for analyzing inconsistencies in a test suite rather than being a generic similarity measure which is not the goal of the test selection methods.

The test suite consistency description by Boroday, Petrenko, and Ulrich [24] is related to the work in this chapter. The paper describes mutual consistency criteria on test cases. In their description, two test cases are inconsistent when the expected SUT outputs for one state of their product are different. In our work, we deal with cases that they call *strongly inconsistent* and define a consistency relation.

As far as tools are concerned, there is a huge and diverse number of software that dynamically analyze code in general. The tools are generically applicable, but none of them concentrate on the quality analysis of tests and its peculiarities and specific properties. In the following, we restrict our tool examples to Java where applicable in order to limit the number of tools to a manageable amount. A well-known and very sophisticated tool for finding memory leaks by simulating the processor is Valgrind [131]. The IBM Rational Purify tool works in a similar way [82]. The jTracert [12] is among the dynamic analysis tools that are meant for program understanding: it generates sequence diagrams from the execution

of Java programs. It does so by monitoring the Java Virtual Machine. Another interesting use is presented by the Daikon tool [48] which generates invariants from its analysis that should hold over the executions it observed. The Runtime Reflection Project [10, 136] allows passive testing of reactive distributed systems by generating runtime monitors from LTL specifications that can be used to monitor arbitrary C++ programs where the C++ code is instrumented. For model checking, probably the best known tools are NuSMV [3], Uppaal [143], and Spin [79]. NuSMV is a symbolic model checker where the input is a network of automata described in the SMV language and the temporal logic is CTL. Uppaal allows analyzing networks of timed automata with binary synchronization. The timed processes can be described using a graphical editor or a textual description. These models can then be simulated or verified. Because the automata in Uppaal are timed, the temporal logic is a timed variant of CTL called *Timed Computation Tree Logic* (TCTL). Finally, Spin uses explicit model checking (as opposed to symbolic model checking) using LTL formulas and communicating EFSMs. The models are described in its own input language called Promela which is an imperative language that looks similar to languages like C. For the communication, the processes may use FIFO queued channels, rendezvous, or shared variables. It allows to simulate the model and to verify it.

All of the mentioned approaches here have in common that they are either generic, i.e., not tailored to any specific language by providing their own abstract input language (as it is the case with model checkers) or they are language-specific for a general-purpose language, i.e., in any case they do not respect the specific properties of test specifications, such as test verdicts, either. Also we do not know any approach that relates its analyses to a specific quality model. Rather, the approaches concentrate on very specific aspects of a quality analysis without relating them to the big picture. Nevertheless, the underlying theory and tools, especially in the model checking area, are very important for our test case analyses and form one important foundation of this work.

5. Applied Automated Test Quality Assessment

So far, we have presented a quality model for test specifications with examples of how to instantiate the model to assess quality characteristics. Subsequently, we introduced methods to analyze property violations in test cases and response inconsistencies in test suites. The presented analysis methods are in fact approaches that are needed to assess certain quality characteristic in a practical instantiation of the quality model. In fact, in Chapter 3.4, we have already presented metrics that require the kind of dynamic analysis that we presented in Chapter 4.4. For example, in Section 3.5.2, we instantiated the *test correctness* subcharacteristic by metrics measuring the number of paths where no test verdict is set (Definition 3.3) and whether test verdicts are set prior to any communicating behavior (Definition 3.4). The methods from Chapter 4.4 allow an effective automated analysis of such metrics where otherwise a manual review would have been required. With this section, we demonstrate how to measure test suites and test cases for an instantiated quality model in an automated manner.

The chapter is divided into three parts: in the first part (Section 5.1), we discuss the underlying implementation used for the measurement and analysis of the test cases. This includes implementations for both the static and dynamic analysis of TTCN-3 test cases. In the second part (Section 5.2), we apply the prototype implementation on TTCN-3 test suites in two different experiments. The first experiment (Section 5.2.2) applies static analysis and test case improvement. We concentrate on the TTCN-3 template structure to first assess and then improve the *changeability* quality subcharacteristic of the quality model for test specifications. The results do not only demonstrate how static analysis is applied in practice to practically instantiate the quality model, but they also indicate how improvements made based on the prior assessment influence the compactness of the test code. In the second experiment (Section 5.2.3), we present and discuss how we applied model-based test case analysis to existing TTCN-3 test cases. The experiment demonstrates the feasibility of the presented approach and provides a discussion on how development guidelines for test suites influence quality characteristics.

5.1. Test Specification Analyzer Implementation

For the experiments performed in this chapter, we have created two software pieces: one performs static analysis and refactoring. In the static analysis, we calculate metrics, ana-

lyze violating thresholds for these metrics, detect patterns within the analyzed syntax tree, and finally, restructure the test code. The other performs dynamic model-based analysis as presented in Chapter 4.4. For the model-based analysis, we have reused the infrastructure created for the first software, but also implemented a behavior simulation engine (or abstract interpreter), which produces event traces that are used for reverse engineering of the behavioral model of the test case under analysis. This model is then verified for specific properties using model checking.

5.1.1. Static Analysis and Refactoring

Our software tool, called *TTCN-3 Refactoring and Metrics Tool* (TRex) [154], provides TTCN-3 tools for the Eclipse Platform [46]. Its underlying infrastructure is powerful and comprises essentially the front-end of a compiler, providing an abstract syntax tree, symbol table, and convenience tools for working with grammatical structures of TTCN-3. Figure 5.1 illustrates the TRex tool chain: the Eclipse Platform provides the basic IDE infrastructure. The TRex components build on top of the Eclipse Platform.

The foundation for most functionality in TRex is the TTCN-3 parser and the resulting syntax tree. For building up the syntax tree for a test suite we use *ANother Tool for Language Recognition* (ANTLR) [117], a parser generator which supports lexing, parsing, and syntax tree creation and traversal. The syntax tree and the symbol table provide the basis upon which most of the functionality is realized, e.g., the metrics and refactoring implementations both use them. As shown in Block (1) of Figure 5.1, the lexer creates a token stream from the TTCN-3 Core Notation which is used by the parser for syntactical validation and for building the syntax tree. The syntax tree is a homogeneous tree data structure using instances of one single class as a node as opposed to a heterogeneous abstract syntax model where each node is represented by a specific class for its type. In addition, the symbol table, also created here, provides additional information for identifiers, such as type information or the syntax tree node of its declaration. As TTCN-3 declarations need no forward declaration, the symbol table is created in a second traversal pass over the syntax tree.

Metrics are collected after the syntax tree for a test suite has been built or updated (Block (3) in Figure 5.1). The metrics are calculated during the traversal of the syntax tree (and its possible dependencies). Simple size metrics are calculated by simply counting the number of occurrences of a syntactical element on tree traversal. Other metrics, such as the cyclomatic complexity, require the creation of a different data structure (such as the intra-procedural control-flow graph) during the traversal or various symbol table lookups, counting or resolving of identifier references and similar. The derived metrics are then processed against analysis rules that are essentially a set of metrics with correspondingly carefully chosen threshold values. Violated analysis rules indicate the possible need for a refactoring. From the user's point of view, markers are displayed in the IDE and associated quick fixes

can be invoked to automatically apply a refactoring that has been associated to the violations of such an analysis rule.

Block (2) in Figure 5.1 depicts how the automated refactorings are realized. On the basis of the static analysis step (Block (1)), workspace resources (i.e., the text files containing the test code) are transformed by means of a programmatic text editor. This programmatic text editor supports operations such as insertion or replacement while tracking text locations to allow multiple subsequent text edits. It is used to weave only the textually changed parts into the original TTCN-3 source files. Therefore, most of the original formatting is preserved. In some cases, an intermediate step involving a syntax tree transformation may become necessary in order to calculate the required changes. In this case, the TTCN-3 core notation to be weaved into the original TTCN-3 source files is obtained by the TRex pretty printer.

A fully automated application of refactorings to code is not always unambiguous. Metrics of the analyzed code change, once a refactoring has been applied. In some cases, the application of multiple refactorings subsequently can lead to the suggestion to reverse a refactoring. Thus, a fully automated assessment and improvement method can lead to loops within the process. In TRex, the refactoring suggestions are applied manually under review of the test developer. Therefore, such loops regarding the application and reversal of refactorings cannot happen.

5.1.2. Model-Based Analysis

The overall tool workflow of the implementation of the model-based analysis of test cases is illustrated in Figure 5.2. The front-end including lexer, parser, syntax tree, symbol table, and various other tools is again performed by the TRex infrastructure. On top of this infrastructure, a new component for the simulation of behavior was implemented. This behavior simulator is a TTCN-3 execution engine as described in Section 4.3.3 and Appendix A.3.

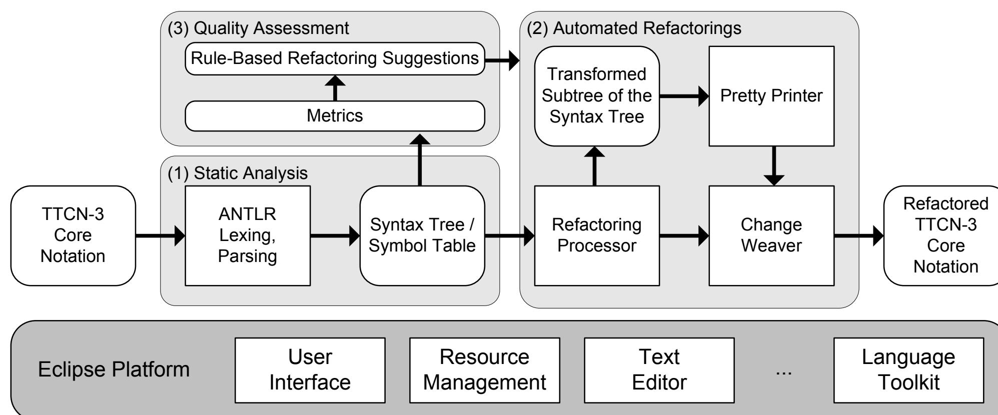


Figure 5.1.: TRex Tool Chain

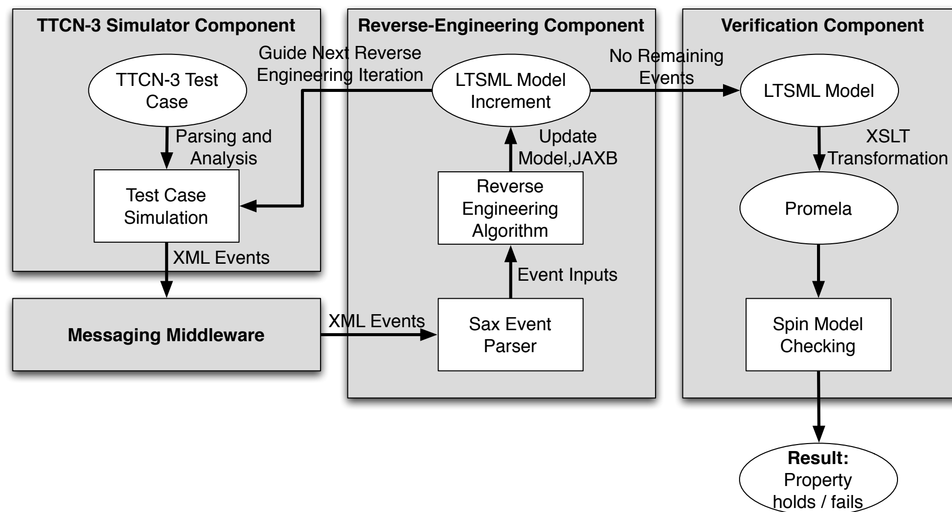


Figure 5.2.: Tool Workflow

It is able to steer the control-flow independently from the actual data values of the execution to cover as many behavioral branches as quickly as possible. The simulator steers into any available behavior in the test case according to a branch coverage strategy. The actual execution is also a tree traversal, where we start in a test case and make sure that at each decision point, only one concrete decision is made. On function, test case, or altstep calls, the simulator remembers its position and jumps to the called structural element. To visit different branches in each simulation iteration, the simulator builds an internal data structure (essentially a graph) that keeps track of the branches that have been visited already to make decisions which not yet visited behavior to execute next. On this graph, we perform the node coloring. In addition to steering the behavior of the test case, the simulator produces the log messages described in Section 4.3.1.

For modeling the properties under analysis (e.g., test verdict events), a strategy for the simulation visitor must be implemented in order to allow the production of the necessary property events. Events such as input and output events are always logged by default. The logged events are passed in an *Extensible Markup Language* (XML) format to a messaging middleware—in the case of this implementation to Apache ActiveMQ [6]. ActiveMQ is an open source message broker implementing the *Java Message Service* (JMS) [114], a *Message Oriented Middleware* (MOM) API. The reasons for using a messaging middleware are manifold. It allows an easy decoupling of the simulator component from the reverse engineering component, easy monitoring of the events, language independence for the consuming components, and the possibility to physically distribute the components in case of scalability problems. Each XML event produced by the simulator is passed individually as one message to the messaging middleware. The reverse engineering component continu-

ously listens for new messages that are received through the messaging middleware. Thus, the architecture allows the processing of the events at the consuming component while the simulation is still in progress, i.e., the simulation and the reverse engineering are actually performed in parallel.

The reverse engineering component consumes each XML event passed by the messaging middleware and parses the events using an XML parser using the *Simple API for XML* (SAX) [100], i.e., the partial XML event strings are directly processed without the construction of a complete syntactical model of the event log in memory. Constructing a *Document Object Model* (DOM) representation would be unnecessary and inefficient as the XML events can be directly processed by the reverse engineering algorithm. The events are processed by the reverse engineering algorithm (see 4.3.2 and Annex A.1) and partial *Labeled Transition System Markup Language* (LTSML) models are created from the processed events. Here, the actual construction of the model takes place by using classes generated from the LTSML XML schema using the XML Binding framework *Java Architecture for XML Binding* (JAXB) [113]. The LTSML model is essentially an XML-based metamodel (*Eclipse Modeling Framework* (EMF) conformant) of the EMIOTS representation. A more detailed presentation of the metamodel can be found in Appendix A.4. Due to the use of JAXB, the actual creation of an LTSML conformant file need not be implemented manually. Rather, by using the JAXB generated classes, the framework is able to serialize the JAXB model automatically to a conformant XML representation. Each sequence of processed log events leads to a partial model that contains the actual execution path of the last simulation iteration, as well as the not yet visited branches that have been colored by the node coloring algorithm to find the path for the next simulation iteration. Thus, the (colored) LTSML model increment is also the input for the next iteration of the behavior simulation.

If the coloring algorithm indicates that all nodes have been visited and no branch is left to be visited, the simulator indicates that it is finished with its branch coverage strategy and that the reverse engineered model is considered to be complete. After that JAXB is used to generate an LTSML conformant textual representation of the LTSML model. For further processing of this textual LTSML representation, multiple *Extensible Stylesheet Language Transformations* (XSLT) style sheets are available. The first XSLT style sheet converts the LTSML representation to the Graphviz Dot [123] format. This allows an easy visualization of the models and the possibility to inspect their correctness during the development of this reverse engineering software system. A second style sheet converts the LTSML model to Promela, the input language of the Spin model checker [79]. The converted Promela model can then be model checked using user-specified LTL formulas, such as those presented in Section 4.4, using the Spin tool. Due to the approach with the intermediate LTSML format, it is easily possible to translate the reverse engineered model to other model checking engines as well: it is just a matter of providing a different style sheet for the different input language.

When we apply the model checker with the appropriate LTL formula to the reverse engineered and then converted model, the output is that either the checked property holds or that it does not hold. If it does not hold, a failing trace is provided. While it is not a part of the prototype implementation, the Promela conversion style sheet and the LTSML model include references to the respective TTCN-3 line numbers. Thus, it would be easy to map the failing trace back to TTCN-3 line numbers to allow an easier analysis of the underlying problem. Details regarding the LTSML metamodel and the LTSML to Promela transformation can be found in [106].

In the following, we present a small example of how the results and intermediate results of the involved steps look like. We begin with a listing of a TTCN-3 module. Listing 5.1 depicts an example test case in TTCN-3 that exhibits a *Fail/Inconc Verdict Decision Before Communication* anomaly (Section 4.4.1.2). As a first step in the test case, a local port *p1* is mapped to a system port. After that, the verdict is initialized with *fail*, i.e., due to the TTCN-3 semantics, the subsequent behavior is not able to overwrite this verdict anymore. This is considered an anomaly since setting a verdict prior to any message exchange or conditional behavior is not a useful test case. After setting the verdict, the test case proceeds by sending a stimulus a through port $p1$. As a response to the stimulus, the test case expects in an alt-statement either a message b on port $p1$, where it tries to set the verdict to *pass* (which does not work in this case), or if any other message is received, the alt-statement is repeated, or if a timeout on timer T occurs, the test case simply terminates by doing nothing.

```
1  testcase test() runs on MyComponent {
2      map(self:p1,system:p1);
3      setverdict(fail);
4      p1.send(a);
5      alt {
6          [] p1.receive(b) {
7              setverdict(pass);
8          }
9          [] p1.receive {
10             repeat;
11         }
12         [] T.timeout {
13         }
14     }
15 }
```

Listing 5.1: Example: Verdict Before Communication (TTCN-3)

After the reverse engineering, we gain an LTSML result model in a textual XML representation. Listing 5.2 illustrates the beginning of the LTSML representation. After the XML header and the reference to its name space, one model called $p0$ is specified. If we had a TTCN-3 test case with parallel behavior distributed on multiple test components, the reverse engineering algorithm would have respectively created multiple models with the *lts* tag in the same file. Following the beginning of the model definition, the initial state of the variables in use is defined. In this case, we only deal with verdict variables

vpass, *vfail*, *vinconc*, *vnone* and a variable *vstimulus* which indicates the first communication event in the test case.

```
1  <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
2  <ltsml xmlns="http://www.trex.informatik.uni-goettingen.de/ltsml">
3      <lts id="p0">
4          <variables>
5              <variable init="false" name="vpass" id="vpass"/>
6              <variable init="false" name="vfail" id="vfail"/>
7              <variable init="false" name="vinconc" id="vinconc"/>
8              <variable init="true" name="vnone" id="vnone"/>
9              <variable init="false" name="vstimulus" id="vstimulus"/>
10         </variables>
```

Listing 5.2: Example: Verdict before Communication (LTSML) - Part 1/6

Following the initial definitions of the available variables of the model, the states are defined (Listing 5.3). Each state is identified by an *id* tag which contains the reference to the model identifier and the state number. State numbers are labeled in an ascending order. States $p0s1$ to $p0s14$ have been omitted from the listing.

```
1          <states>
2              <state id="p0s0"/>
3              <!-- ... -->
4              <state id="p0s15"/>
5          </states>
```

Listing 5.3: Example: Verdict before Communication (LTSML) - Part 2/6

Following the state definitions, the actions are defined (Listing 5.4). An action may have the type *input*, *output*, or *internal*, which refers to the partitioning of the action set. The identifiers for the actions are labeled analogously to the states. Furthermore, they may have descriptions which essentially correspond to the labels in the LTS definition. Here, the respectively processed TTCN-3 statements are mapped to the labels. Transitions may also alter the set of variables. This is indicated by the *changeVariable* blocks that assign new values to the variables. The *varref* attribute references the respective variable by its identifier. Input and output actions additionally carry the attribute *portRef*, in which a channel identifier must be provided.

```
1          <actions>
2              <action type="internal" id="p0a0"><description>map(self:p1, system:p1)</description></action>
3              <action type="internal" id="p0a1"><description>setverdict(fail)</description></action>
4              <action type="internal" id="p0a2"><description>varaction</description>
5                  <changeVariables>
6                      <changeVariable value="false" varRef="vpass"/>
7                      <changeVariable value="true" varRef="vfail"/>
8                      <changeVariable value="false" varRef="vinconc"/>
9                      <changeVariable value="false" varRef="vnone"/>
10                 </changeVariables>
11             </action>
12             <action type="internal" id="p0a3"><description>p1.send(a)</description></action>
```

```
13            <action type="internal" id="p0a4"><description>varaction</description>
14                <changeVariables>
15                    <changeVariable value="true" varRef="vstimulus"/>
16                </changeVariables>
17            </action>
18            <action type="output" portRef="0" id="p0a5"><description>a</description></action>
19            <action type="internal" id="p0a6"><description>T.timeout</description></action>
20            <action type="tau" id="p0a7"><description>tau</description></action>
21            <action type="input" portRef="0" id="p0a8"><description>b</description></action>
22            <action type="internal" id="p0a9"><description>setverdict(pass)</description></action>
23            <action type="internal" id="p0a10"><description>varaction</description>
24                <changeVariables>
25                    <changeVariable value="true" varRef="vpass"/>
26                    <changeVariable value="false" varRef="vfail"/>
27                    <changeVariable value="false" varRef="vinconc"/>
28                    <changeVariable value="false" varRef="vnone"/>
29                </changeVariables>
30            </action>
31            <action type="input" portRef="0" id="p0a11"><description>*</description></action>
32            <action type="internal" id="p0a12"><description>repeat</description></action>
33        </actions>
```

Listing 5.4: Example: Verdict before Communication (LTSML) - Part 3/6

Following the action definitions, ports are defined (Listing 5.5). They are numbered in an ascending order. If two ports in a multiple-model definition carry the same identifier, the corresponding transformation style sheets associates them with a channel, i.e., the test configuration is static.

```
1        <ports>
2            <port id="0"><description>0</description></port>
3        </ports>
```

Listing 5.5: Example: Verdict before Communication (LTSML) - Part 4/6

The transition definitions (Listing 5.6) are defined by specifying attributes for referencing the source state, the target state, and the corresponding action. The transitions themselves again carry an identifier with a naming scheme, similar to states and transitions.

```
1        <transitions>
2            <transition targetRef="p0s1" sourceRef="p0s0" actionRef="p0a0" id="p0t0"/>
3            <transition targetRef="p0s2" sourceRef="p0s1" actionRef="p0a1" id="p0t1"/>
4            <transition targetRef="p0s3" sourceRef="p0s2" actionRef="p0a2" id="p0t2"/>
5            <transition targetRef="p0s4" sourceRef="p0s3" actionRef="p0a3" id="p0t3"/>
6            <transition targetRef="p0s5" sourceRef="p0s4" actionRef="p0a4" id="p0t4"/>
7            <transition targetRef="p0s6" sourceRef="p0s5" actionRef="p0a5" id="p0t5"/>
8            <transition targetRef="p0s7" sourceRef="p0s6" actionRef="p0a6" id="p0t6"/>
9            <transition targetRef="p0s8" sourceRef="p0s7" actionRef="p0a7" id="p0t7"/>
10           <transition targetRef="p0s9" sourceRef="p0s8" actionRef="p0a7" id="p0t8"/>
11           <transition targetRef="p0s10" sourceRef="p0s9" actionRef="p0a7" id="p0t9"/>
12           <transition targetRef="p0s11" sourceRef="p0s6" actionRef="p0a8" id="p0t10"/>
13           <transition targetRef="p0s12" sourceRef="p0s11" actionRef="p0a9" id="p0t11"/>
14           <transition targetRef="p0s13" sourceRef="p0s12" actionRef="p0a10" id="p0t12"/>
15           <transition targetRef="p0s8" sourceRef="p0s13" actionRef="p0a7" id="p0t13"/>
16           <transition targetRef="p0s14" sourceRef="p0s6" actionRef="p0a11" id="p0t14"/>
17           <transition targetRef="p0s15" sourceRef="p0s14" actionRef="p0a12" id="p0t15"/>
18           <transition targetRef="p0s6" sourceRef="p0s15" actionRef="p0a7" id="p0t16"/>
19       </transitions>
```

Listing 5.6: Example: Verdict before Communication (LTSML) - Part 5/6

The LTSML model definition concludes by providing a start state and possible end states (in case we need valid end states for deadlock analyses). Both states are defined by providing a reference to their respective state in the set of states. Listing 5.7 depicts this last part of the LTSML definition.

```
1         <startState stateRef="p0s0"/>
2         <endStates>
3             <endState stateRef="p0s10"/>
4         </endStates>
5     </lts>
6 </ltsml>
```

Listing 5.7: Example: Verdict before Communication (LTSML) - Part 6/6

Based on the LTSML definition, we can apply XSLT style sheets to visualize the model and to transform it to an input language of a model checker—in our case Promela. Figure 5.3 illustrates the resulting Dot model when transforming the LTSML representation to a Graphviz digraph. Note that we do not provide the textual presentation of the transformed Dot input notation here.

The Promela transformation yields the result in Listings 5.8, 5.9, and 5.10. The transformation to Promela is not entirely straightforward. A question is how to deal with communicating events between the SUT and among the test components. Promela supports message based communication among its processes. However, we do not have the behavior of the SUT specified in the TTCN-3 test case and also do not want to model it. The answer to this problem is to map SUT communication to internal events while we preserve the message-based paradigm among the communication between internal test components. We can differentiate SUT ports from internal ports by checking whether ports with the same identifiers exist in the respective model. If there are matching identifiers, the channel is internal between test components. Otherwise, it is a channel to the SUT. As there is no communication among multiple test components in the example, all message-based communication is treated as SUT communication.

The Promela model starts by defining internal actions (Listing 5.8). We model these internal actions by means of predefined integer values within the Promela code that are replaced using the C pre-processor. Normally, the *mtype* types in Promela would be more appropriate for this purpose. However, there is an upper limit of 255 to the number of *mtype* definitions. Therefore, we map internal actions directly to integer values which allow a considerably higher number of internal actions.

```
1  #define p0a0_map___self_p1__system_p1__ 1
2  #define p0a1_setverdict___fail__ 2
3  #define p0a2_varaction 3
4  #define p0a3_p1_send___a__ 4
5  #define p0a5_a 6
6  #define p0a6_T_timeout 7
7  #define p0a7_tau 8
8  #define p0a8_b 9
9  #define p0a9_setverdict___pass__ 10
10 #define p0a11__ 12
11 #define p0a12_repeat 13
```

Listing 5.8: Example: Verdict before Communication (Promela) - Part 1/3

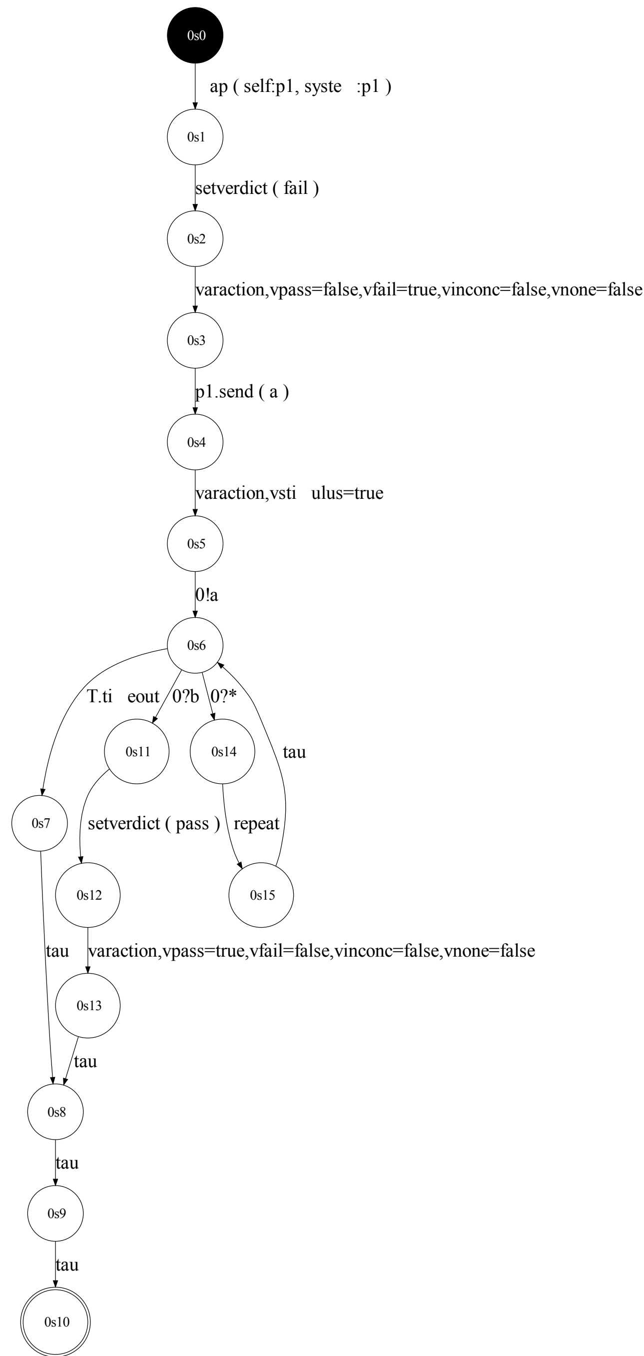


Figure 5.3.: Example: Verdict before Communication (Dot Visualization)

Listing 5.9 provides the necessary Promela code before the actual *proctype* definition is provided. This includes the channel *ch*0 corresponding to the channels defined in LTSML. The transformation style sheet defaults to a queue length of two to limit the complexity of the verification. Subsequently, we define the variables. These definitions directly match the variable definitions in the LTSML model. However, the variables in Promela are global over all *proctypes*, whereas in LTSML the variables are local to the model. Finally, we define integer guard variables for each transition in the model. These guard variables are used as a workaround to problem that occurs due to missing end conditions in the Promela model (more details on this workaround can be found in the paragraph below).

```
1  chan ch0 = [2] of {int};
2
3  /* Variables */
4  bool vpass = false;
5  bool vfail = false;
6  bool vinconc = false;
7  bool vnone = true;
8  bool vstimulus = false;
9
10 /* Transition Guard Variables (to disable unwanted acceptance cycles)*/
11 int p0t0 = 0;
12 // ...
13 int p0t16 = 0;
```

Listing 5.9: Example: Verdict before Communication (Promela) - Part 2/3

Listing 5.10 depicts the actual process definition of the test case behavior. The behavior is modeled using *label* and *goto* statements. While such constructs are considered to be problematic in general-purpose code, the Promela code in generated and will never be maintained by a developer. Therefore, the use of these constructs does not present any problem. Each state is marked by a label. The end state is mapped to the special label *end*. The actual transitions are defined by the behavior following the state label. States with only one outgoing transition provide their action (mostly by means of simply specifying the predefined integer value that is defined using the pre-processor). If a state has multiple outgoing transitions, an *if-statement* in Promela follows. The actions defined in the transitions then take place after the guard.

The provided guards are the only unintuitive mapping of the LTSML model to Promela. The reason for this are acceptance cycles. For example, in the TTCN-3 behavior, the repeat statement creates a cycle within the behavior and, in fact, no proper termination criterion is provided within this behavior. This is the case, since in this kind of message-based communication, we simply expect that either another message arrives or a timeout occurs. However, the Promela model would not know when a message from an SUT can be expected or not—as previously stated, we do not model the SUT. Also, we do not model time so that all our models are untimed. As a result, the verification engine of Spin would detect an acceptance cycle for the repeat statement, i.e., there is the possibility that the behavior gets stuck and always chooses the branch where any message other than message *b* is received on *p*1. There is no easy way to work around this problem. To trick this behavior of Spin, we have

inserted counters for each transition as guards which are non-deterministically incremented if the transition is chosen. When it is incremented, the guard is blocked. However, it can be released non-deterministically again at any time when the artificial else branch of the same condition is chosen where the counters are set back to zero. Using this workaround, we avoid the problem with the acceptance cycles, while not limiting the behavior in an invalid way. However, such cycles generally create infinite behaviors and so does the workaround. If such cycles occur, the model checking engine has to terminate to some search depth without looking further. This is not a problem due to the workaround, but it is inherent to the behavior that is analyzed.

Finally, if variables are altered, the alterations are realized in an *atomic* block to avoid property violations within the same transition.

```
1  active proctype p0(){
2  start: goto p0s0;
3   p0s0: p0a0_map___self_p1__system_p1__;
4          goto p0s1;
5   p0s1: p0a1_setverdict___fail__;
6          goto p0s2;
7   p0s2: atomic { vpass = false; vfail = true; vinconc = false; vnone = false; };
8          p0a2_varaction;
9          goto p0s3;
10  p0s3: p0a3_p1_send___a__;
11         goto p0s4;
12  p0s4: atomic { vstimulus = true; };
13         p0a2_varaction;
14         goto p0s5;
15  p0s5: p0a5_a;
16         goto p0s6;
17  p0s6: if
18         :: p0t6 < 1 -> if :: p0t6++; :: else -> skip; fi; p0a6_T_timeout; goto p0s7;
19         :: p0t10 < 1 -> if :: p0t10++; :: else -> skip; fi; p0a8_b; goto p0s11;
20         :: p0t14 < 1 -> if :: p0t14++; :: else -> skip; fi; p0a11__; goto p0s14;
21         :: else ->
22               p0t6=0;
23               p0t10=0;
24               p0t14=0;
25               goto p0s6;
26         fi;
27  p0s7: p0a7_tau;
28         goto p0s8;
29  p0s8: p0a7_tau;
30         goto p0s9;
31  p0s9: p0a7_tau;
32         goto p0s10;
33 p0s10: goto end; /* end state */
34 p0s11: p0a9_setverdict___pass__;
35         goto p0s12;
36 p0s12: atomic { vpass = true; vfail = false; vinconc = false; vnone = false; };
37         p0a2_varaction;
38         goto p0s13;
39 p0s13: p0a7_tau;
40         goto p0s8;
41 p0s14: p0a12_repeat;
42         goto p0s15;
```

```
43  p0s15: p0a7_tau;
44          goto p0s6;
45    end: skip;
46  }
```

Listing 5.10: Example: Verdict before Communication (Promela) - Part 3/3

Having seen the transformations for a small example in detail, we now present case studies applying static and dynamic analysis techniques to test specifications.

5.2. Case Study

In the following, we present the experiments that we have performed with our implementations for the static and dynamic analysis for TTCN-3 test cases. We start this section by giving a description of the test suites that were subject of the analyses.

5.2.1. The ETSI SIP, IPv6, and HiperMAN Test Suites

The experiments in this chapter have been performed primarily on three *Abstract Test Suite*s (ATSs) that have been developed by ETSI, i.e., the test suites originate from standardization.

The *Session Initiation Protocol* (SIP) ATS [50] is a conformance test specification for the *Internet Engineering Task Force* (IETF) *Request for Comments* (RFC) 3261 [125]. It is the basis for the conformance testing of SIP equipment. The ATS is designed in such a way that equipment passing this conformance test should have a higher probability of interoperability, in particular between SIP equipment of different manufacturers. To achieve that, the test suite takes multiple roles: the user agent, registrar in the outbound proxy, or the registrar in the redirect server.

The *Internet Protocol Version 6* (IPv6) core protocol ATS [51] is a conformance test specification for the core function of the Internet Protocol, version 6 as defined in IETF RFCs 1981 and 3513 [76, 98]. It is also the purpose of this ATS to ensure a higher probability of interoperability between IPv6 equipment from different manufacturers. The ATS tests hosts and routers. Either a one-to-one connection between the tester and the SUT is established or the SUT is connected to two PTCs that act as router and host respectively.

The *Worldwide Interoperability for Microwave Access* (WiMAX)/*High Performance Metropolitan Area Network* (HiperMAN) Subscriber Station ATS [49] is a conformance test specification to ensure the interoperability of *Broadband Radio Access Network* (BRAN) HiperMAN/WiMAX equipment from different manufacturers over the air, according to the ETSI standards on the HiperMAN Data Link Control Layer and System Profiles [53, 54] and IEEE standards 802.16-2004 and 802.16e-2005 [83, 84]. The IEEE 802.16-2004 standard and the ETSI HiperMAN specification are compatible due to the fact that there is a mapping between the ETSI specifications and corresponding IEEE components. The ATS tests the *Data Link Control* (DLC) layer of the protocol, in particular the subscriber station and the base station.

All three specifications follow the ISO standard for the methodology of conformance testing ISO/IEC 9646 [85], as well as ETSI rules for conformance testing [52] as basis for their test methodology. Some basic size measures of these test suites can be found in Table 5.1.

	SIP 4.2.5	SIP 4.1.1	IPv6 1.1	HiperMAN 2.2.1
Lines of code	61989	61282	41801	22803
Number of templates	381	383	148	456
Number of test cases	608	609	286	72

Table 5.1.: Size of ETSI Test Suites

5.2.2. Static Assessment and Improvement

The motivation for this experiment is the observation that, especially in TTCN-3 test suites, test data definitions play an important role and present a significant part of what needs to be maintained. Hence, in this study, we concentrate on the *maintainability* quality aspect—in particular the *changeability* subcharacteristic—of TTCN-3 templates by (1) detecting opportunities to avoid changeability issues in templates and (2) by applying automatic refactoring to restructure the templates in order to avoid the detected issues and improve the quality for the changeability subcharacteristic.

For assessing the quality of TTCN-3 test suites in terms of analyzability and changeability, we have chosen metrics from our exemplary quality model instantiation in Chapter 3.5.2. For that purpose, we measure those metrics that relate to templates and that have been derived from applying the GQM approach:

- Removable Definition (Definition 3.12).
- Similar Template (Definition 3.13).

The refactorings are applied according to the following principle rules:

- A template definition that is not referenced (number of references to a template = 0) should be removed.
- A template definition that is referenced once (number of references to a template = 1) should be inlined and its definition should be removed (application of the *Inline Template* refactoring that, for parameterized templates, includes the inlining of parameters).
- If two or more template definitions exist for the same type and template values differ for the same template fields and these differing fields account for a percentage of at

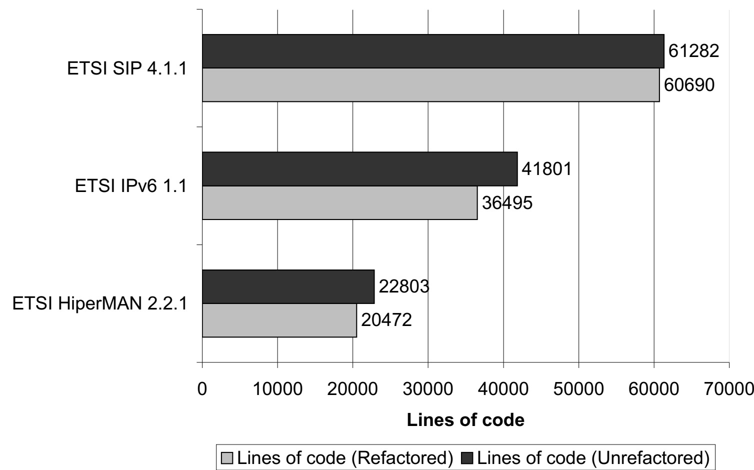


Figure 5.4.: LOC Before and After Applying Refactorings

least 30% of the overall fields of the template definition, then the templates should be reduced to a single parameterized definition (application of the *Parameterize Template* refactoring).

By removing unused template definitions and inlining singular referenced ones, code clutter is reduced. By merging templates, changeability is enhanced due to greater flexibility and less duplication. By reducing the volume of the test suite source code through parameterization, the analyzability quality subcharacteristic may also benefit.

The charts in Figures 5.4 and 5.5 visualize the effect of these refactorings on the ATSs in terms of the *Number of lines of TTCN-3 source code* metric (physical lines of code) and the *Number of templates* metric. The measurements indicate that results certainly depend on how much a test suite has already been optimized with respect to the factors mentioned. For the SIP ATS, for example, the effect is less noticeable—especially in terms of lines of code, since the template definitions do not constitute the majority of the test suite volume. The effect on the IPv6 ATS on the other hand is clearly visible. The number of template definitions could be reduced to less than half the original number so that more than 5000 lines of code could be saved. It should be noted that the number of inlined templates can be neglected in this case. The impact originates from removing unused and merging similar or duplicate templates. The WiMAX/HiperMAN ATS yields a similar result. The number of templates could be reduced by approximately a third and the test suite size could be reduced by more than 2000 lines of code.

When taking into account that these are the results of merely three refactoring rules which have been applied, a higher number of implemented rules and refactorings, also supporting behavioral and structural quality aspects, is likely to have an even more noticeable impact on the test suite source code.

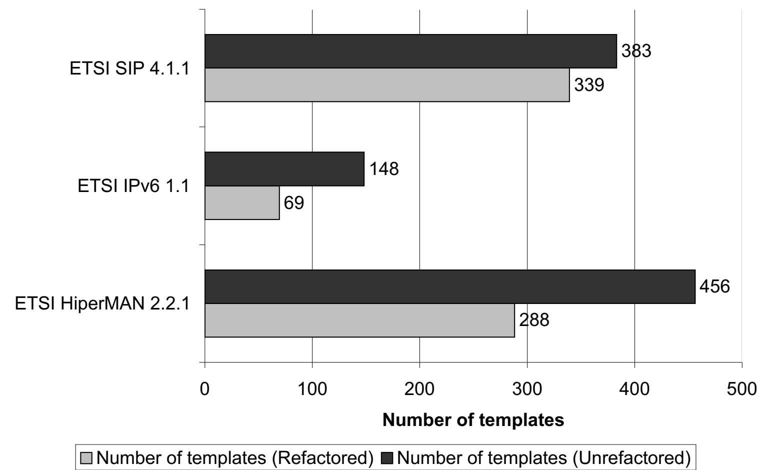


Figure 5.5.: Number of Templates Before and After Applying Refactorings

5.2.3. Model-Based Assessment of Dynamically Detectable Properties

In the second case study, we performed dynamic analyses as presented in Chapter 4.4 to assess quality attributes of an industrial-size test suite. Subject of this experiment is the ETSI conformance test specification for SIP [50]. The goal of the experiment was comprised by the following items:

- To show that an automated dynamic analysis is practically feasible and applicable to test cases of a real-world test suite.
- To find an indication regarding the precision of the analysis results.
- To detect possible anomalies in the analyzed test cases.

For these purposes, we chose to analyze aspects of the *reliability*, *compliance*, and *test correctness* quality attributes of the quality model for test specifications. Following the GQM approach and the exemplary quality model instantiation of Chapter 3.4, we evaluated the reliability subcharacteristic with the following metric:

- Timeout Inconsistency (Definition 3.8).

For the compliance subcharacteristic, we used the following metric:

- Verdict/Timer Inconsistency Degree (Definition 3.22).

For the test completeness subcharacteristic, we used the following metrics:

- Test Verdict Completeness (Definition 3.3).
- Early Test Verdict (Definition 3.4).

For conducting the case study, we selected a subset of the SIP ATS (version 4.1.1 unless stated otherwise) that on the one hand covered different functional areas of the test suite and on the other hand also test cases which are variants of others. Table 5.2 lists the analyzed test cases with some basic metrics about their reverse engineered models to indicate their respective sizes. The goal of this particular experiment was not only to find anomalies in this test suite, but also to check the recall and precision of the analysis approach presented in this thesis using a verifiable small-scale example. Recall and precision are terms from information retrieval theory [146]: let R be the set of relevant hits and P be the set of positively identified hits. *Recall* is defined as $\frac{|R \cap P|}{|R|}$ and can empirically be interpreted as the probability that an anomaly present in a test case will be detected. *Precision* is defined as $\frac{|R \cap P|}{|P|}$ and can empirically be interpreted as the probability that a detected anomaly is a real anomaly (and not a false positive). We evaluated recall and precision as follows: from our test case set, we considered P to be the set of test cases that have been identified to contain an anomaly by our automated anomaly detection software prototype. The set R is then the set of test cases that in fact contained an anomaly as identified by manual verification performed by a human reviewer. While the model checker delivers failure traces to identify where an anomaly may occur, we did not use these traces to measure the precision regarding false failure traces. The reverse engineering algorithm must perform some abstractions on the TTCN-3 specification in order to fit the TTCN-3 test behavior into the formal model. Therefore, we anticipate the possibility of false positives in the delivered failure traces.

5.2.3.1. Test Reliability Assessment

To exemplarily assess the *test reliability*, we chose to analyze the test cases for paths in which a timer is started and where subsequently a timeout event on the timer is caught by referencing the timer directly, while at the same time the test case contains a path where the timer is caught implicitly using the *any timer.timeout* statement. Such *any timer.timeout* branches are often found in activated defaults. This is considered a possible anomaly as a directly referenced timer is superfluous unless the follow-up behavior is different from the follow-up behavior of the alt-branch in the default. We consider this anomaly analysis to be a rather project-specific than generic. Therefore, we explain the analysis method here rather than in Chapter 4.4.

The actual analysis setup was to some extent different from the generic properties that we have presented in Chapter 4.4. The properties we have presented so far verify behavior that should hold in all executions (*desired behavior*). For this analysis, it is easier to express the properties in the form of *error behavior*, i.e., we specify exactly what must not happen in any path rather than what must hold in all paths.

For the actual analysis, we ran two model checking passes: the first pass checked whether there exists a path in which a timer is started followed by a timeout event on the timer which is caught by an *any timer.timeout* statement. The second pass checked whether there exists a path in which a timer is started and then caught by a directly referenced timer. Intuitively, we

Test case	Number of States	Number of Actions	Number of Transitions
SIP_CC_OE_CE_TI_001	290	106	330
SIP_CC_OE_CE_TI_006	238	110	280
SIP_CC_OE_CE_TI_008	246	107	291
SIP_CC_TE_CR_V_016	3060	247	3201
SIP_MG_RT_I_001	117	75	137
SIP_MG_RT_I_003	141	86	169
SIP_MG_RT_I_005	114	72	135
SIP_MG_RT_V_001	131	83	155
SIP_MG_RT_V_002	131	83	153
SIP_MG_RT_V_008	466	158	564
SIP_QC_OE_TI_001	185	77	209
SIP_QC_OE_TI_005	184	78	208
SIP_QC_OE_TI_007	165	73	184
SIP_QC_OE_V_002	212	107	251
SIP_QC_OE_V_004	205	103	241
SIP_QC_OE_V_009	185	98	218
SIP_RG_RT_TI_001	143	73	164
SIP_RG_RT_TI_004	153	67	174
SIP_RG_RT_TI_007	153	68	175
SIP_RG_RT_V_001	100	69	118
SIP_RG_RT_V_002	115	71	136
SIP_RG_RT_V_016	100	69	118

Table 5.2.: The Analyzed Test Cases

can express this with formulas such as the following (corresponding to the global *existence* specification pattern for the timer start—timer timeout sequence):

- $\Diamond\,(timerStart\ \mathcal{U}\ anyTimerTimeout)$, and
- $\Diamond\,(timerStart\ \mathcal{U}\ directTimerTimeout)$.

However, we need to extend these formulas to consider the respective timer analysis blocks. After all, we need to make sure that we analyze blocks that are comparable for both passes. Otherwise, the model checker result might refer to different alt-blocks in both passes. To ensure this, we apply a variant of the *Existence* property pattern. The required variant checks whether a P becomes true after a property Q becomes true:

$$\Box\,(\neg\, Q) \vee \Diamond\,(Q\ \wedge\ \Diamond P) \tag{5.2.1}$$

The variables P and Q are then assigned as follows:

- The variable P is $timerStart\ \mathcal{U}\ anyTimerTimeout$ for the first pass.
- The variable P is $timerStart\ \mathcal{U}\ directTimerTimeout$ for the second pass.
- Q is the respective enabling event for the analysis block.

The model checking then verifies whether this property is violated: if a violation occurs, there exists a path in the analysis block where a timer is started and then timeouts either due to *any timer* or a direct timer reference. We deal with a possible inconsistency if both the first and the second model checking pass deliver a property violation as a result. We analyzed the first behavioral block of the selected test cases for such timer anomalies to allow a manageable verification by hand. Performing the automatic reverse engineering and verification with our prototype tool yielded that all of our selected test cases contained this possible anomaly. As a result, the test developers would have to check by hand whether this behavior is intended or whether they possibly missed the any timer timeout-branch due to the implicitly attached default behavior which is easy to overlook. The result also matched our manually performed analysis, i.e., all blocks where this situation happens had been identified by the tool and all positively identified blocks were in fact blocks that exhibited the anomaly.

5.2.3.2. Compliance Assessment

As stated above, we investigated if any non-timeout paths in the test suite where a verdict would be set before a corresponding stop statement of a timer existed. To set verdicts after a timer has been stopped is a behavioral pattern found in the ETSI test suite that we have identified while manually inspecting it. Like the reliability assessment before, we consider this analysis to be rather project-specific and, therefore, explain the analysis here instead of Chapter 4.4. To express this property, we have to cover the following two conditions which have to apply for paths in which a verdict is set:

- A timeout variable is set to true.
- A timer stop variable is set to true which has been started before.

This can be expressed by the *Precedence* specification pattern. In our case, the pattern variant *S precedes P before Q* applies which is represented by the following LTL formula:

$$\Box \neg Q \vee \Diamond(Q \wedge ((\Box \neg P) \vee (\neg P\ \mathcal{U}\ S))) \tag{5.2.2}$$

We model our situation with the following variables:

- The variable *tstart* indicates whether a timer has been started.
- The variable *tstop* indicates whether this timer has been stopped.
- The variable *ttimeout* indicates whether this timer had a timeout.
- The variables *vpass*, *vfail*, *vinconc*, and *vnone* represent their respective test verdicts.

For the analysis, we assign S, P, and Q to the following variable expressions:

- S is represented by tstop $\vee$ ttimeout.
- P is represented by vpass $\vee$ vfail $\vee$ vinconc.
- Q is represented by tstart.

In natural language, we therefore check whether a timer stop statement or a timer timeout precedes a verdict change to any verdict (except for none) if the corresponding timer has been started.

We applied this analysis to both version 4.1.1 and version 4.2.5 of the SIP ATS targeting the respective timers that are active within the test behavior of the respective tests with the applied formula. While we did not find any anomaly in our selected test case subset of version 4.1.1, we identified the test case *SIP_CC_TE_CR_V_016* as test case in version 4.2.5 which had the setverdict statement switched with the timer stop statement. A manual inspection of the TTCN-3 test case confirmed that this pattern violation is indeed the case and a text file differencing tool also confirmed that this test case had been changed in this subsequent version of the SIP ATS. We did not have any false positives and a manual inspection of the other test cases also yielded that the anomaly only happens in the test case that we detected.

5.2.3.3. Test Correctness Assessment

Our measurement of the test correctness characteristic consisted of two separate subexperiments. The first experiment consisted of measuring whether any paths with missing verdicts or verdicts set before any kind of communication with the SUT using the property descriptions from Sections 4.4.1.1 and 4.4.1.2 existed. All 21 test cases were handchecked for these properties before applying the prototype tool for the recall and precision assessment.

Interestingly, the result of the manual analysis for paths with problematic test verdicts in the SIP ATS yielded the result that none of the checked test cases in fact exhibited the two analyzed anomalies. In all the selected test cases (and additional hand-picked sample test cases of the SIP ATS) were no cases where verdicts would not be set or set before any communication or trigger. The tool reported no false positive test cases and, therefore, detected correctly that these anomalies did not exist.

After manually checking other test cases from the SIP ATS as well as a number of test cases from other ATSs from ETSI, we determined that it is not useful to search for violations of the missing verdict property in test suites that probably had been reviewed and validated for such anomalies. Also, going back in the version history of the same test suite did not exhibit new problems. Most changes regarding corresponding test cases involved the renaming of identifiers, inlining of behavior from elsewhere, adding new behavior, refinements regarding the logging, or other editorial-type changes that do not drastically change the semantics of the behavior.

As a result, the new experiment was to mutate the same subset of test cases from the first experiment regarding paths with missing verdicts. The mutation of these test cases itself was performed by third party researchers with a good degree of TTCN-3 knowledge. They hand-picked an arbitrary number of test cases among the test suite subset and changed their behavior in such a way that the missing verdict property is violated. After that, they handed over the modified test suite to the persons running the model-based analysis implementation on the test suite without telling them what has been altered and what the respective effects are (i.e., a double-blind experiment in the sense that neither the implementation nor the researcher running the analysis implementation know what the correct outcome should be). After applying the tool, a set of test cases with missing verdicts and a set of test cases with no missing verdicts were identified and compared against the notes that the researcher who mutated the test cases made. All altered test cases that affect the verdicts were positively identified. In addition, no false positives were reported in the sense that the implementation detected a test case with a missing verdict where no such path was present. Consequently, our experiment yielded perfect recall and precision.

It is interesting to note that the number of failing traces provided by the model checker indicated for some test cases an infinite amount of paths that violate the properties. The numbers provided by the model checker are absolute, but the model checker limits the number of search paths using an upper bound for the amount of memory that it may use. Therefore, the interpretation is that the model checker had to terminate the verification. The reason for these infinite amounts of failing traces is the fact that the verified model is an abstraction. For the same reason, we did not bother to manually identify the number of paths that lead to a missing test verdict in the mutated test cases—in many cases, the actual number of paths is infinite.

5.2.3.4. Conclusions

The value of the experiments and their empirical meaning is limited by their rather small scale. As the reverse engineered models are designed to be logically sound, we expect the recall to be perfect. Nevertheless, the performed experiments do not exhibit the weakness of our approach regarding the fact that the method is not logically complete—a circumstance that exists due to our negligence of various data manipulations in the TTCN-3 behavior. This fact may be a weak indicator for the hypothesis that black-box test cases such as the SIP ATS are primarily stimulating the SUT and evaluating the incoming messages rather than dealing with a huge amount of data manipulations. As a result, the experiments confirm the intuition that data abstractions in test cases have rather weak repercussions when model checking test case behavior.

The overall effort for automatically checking a test case is relatively low. The test cases verified using the automated analysis tool were always reverse engineered and model checked in less than 30 seconds each (modern dual-core machine with 3Ghz and 4GB of RAM)—depending on the test cases analyzed. This fact also reinforces the intuition that

the behavioral test case complexity is often rather low and well suited for automatic verification. The performed experiments suggest that automatic model-based test case analysis using model checking is in fact a feasible way to check specific properties over test cases in a test suite. On the one hand, the method positively identifies test cases candidates that violate a property and thus may exhibit the problem under analysis. On the other hand, the computational effort and time needed to apply the prototype implementation is low enough to be practical.

The fact that the analyzed parts of the ETSI SIP test suite did not expose any anomalies regarding the verdicts, however, provokes a discussion for its reasons. The ETSI test suites are developed using a set of guidelines. The technical specification of the SIP conformance test suite itself formally only defines naming conventions and implementation conventions. The naming conventions specify how type definitions, template definitions, constants, and other TTCN-3 constructs must be named. The implementation conventions specify how elements of the SIP protocol have been mapped to TTCN-3 definitions and how they are supposed to be used. Nevertheless, the behavior itself exhibits repeating patterns how the test case behavior is constructed. Examples for such patterns are the following:

- Following the variable declarations, the ports are mapped and connected first.
- Following the port connections and mappings, one single default is activated that receives any message on the respective port that is subject of the test case and that catches timeouts of any timer. In both cases, the test verdict is defined as a failing verdict and the test case is terminated.
- There is only one default activated in the test case behavior and defaults are only activated in test cases and not in functions or altsteps.
- In each branch of an altstep, the first thing that is done after possible variable declarations is the stopping of timers.
- Every alt-branch of test case behavior sets a test verdict after the timer has been stopped.
- In a timeout-branch, the timers are not explicitly stopped and the test verdict can be set independently from a stop timer statement.
- If the alt-branch contains a repeat statement, there may not be setverdict statements.
- After a verdict has been set to fail, the test case is terminated using the stop statement.

These are just a few examples for guideline patterns that have been followed in the development of the test suite. Applying such guidelines in the development of a test suite prevents problems such as the missing verdict anomaly from happening. In this concrete case, the rule that each alt-branch in a test case must set of test verdict prevents the missing verdict anomaly from happening. A test developer who strictly adopts these rules produces considerably less anomalies of the kinds that we have presented before. The case study therefore nicely demonstrates the value of constructive quality assurance in contrast to analytical quality assurance: while we identified the anomalies after mutating the test suites

and therewith validated that our approach works, the value of the analytical approach to quality assurance is most apparent when the constructive quality assurance lacks, for example, when test specifications are developed ad-hoc without such a set of strict guidelines or when the test developers are inexperienced or non-attentive.

Another observation is that the enforcement of such guidelines can also be checked by tools. In contrast to our approach to dynamic analysis, the tool-based enforcement of such guidelines is often less complex as the rules have a rather local scope. Thus, they often require only static analysis. Nevertheless, the model-based approach to dynamically analyze test cases is a commendable additional measure to check certain properties that may have slipped through despite the utilization of constructive measures such as guidelines and guideline checkers.

6. Conclusion

In this last chapter, we summarize the thesis and its contributions (Section 6.1). Beyond that, we state possible research items which extend or refine the results and methods presented in this thesis (Section 6.2).

6.1. Summary

The general purpose of this work was to show that *systematic machine-based analytical quality assurance using both static testing and dynamic testing respecting test-specific characteristics is possible and feasible*. To allow a systematic analytical quality assurance, we first defined a quality model for test specifications based on the ISO/IEC 9126 quality model. The presented quality model provides a framework in which analytical quality assurance takes place and describes characteristics and subcharacteristics relevant for the quality of abstract test specifications. We as well described how to instantiate this quality model for a concrete project and a concrete language, in our case TTCN-3. To develop appropriate quality metrics, we suggested the application of the GQM method. This method ensures the modeling of goal-oriented metrics, guaranteeing that each metric measure has a meaning for the assessment at hand.

Having established a framework for classifying test specification quality, we presented a method for the dynamic analysis of test cases. This included a reverse engineering algorithm and a number of generic, test-specific, and TTCN-3-specific properties. The properties are presented in the form of a catalog and the analysis is expressed in the form of temporal logic formulas in LTL.

The dynamic test case analysis followed the response inconsistency analysis which takes a more global view on the test suite. With the help of this method, we were able to detect test cases which are inconsistent regarding their responses to equal stimuli prefixes. After an introductory example, we presented the **reco** relation, which defines response consistency for two test case models. Moreover, we presented an analysis algorithm and discussed a few scenarios for this relation that are not self-explanatory.

Finally, we have validated our approach to analytical quality assurance in a case study. We have measured quality attributes of an industrial-size standardized test specification using static analysis and dynamic analysis. In one experiment, we have measured bad smells in test specification, i.e., locations in the test specification presenting possible problems, and have associated refactorings to them. This allowed the automatic restructuring of the test

suite. The results indicated that a noticeable effect can be achieved already with a small number of metrics and refactorings. The other experiment, in which we performed model-based analysis, has validated that our approach to the dynamic analysis of test specifications is feasible in practice and that the amount of false positives is low.

6.2. Outlook

The work of this thesis presents a first step towards the actual application of systematic analytical quality assurance for test specifications. We have shown that the methods presented in this thesis are practically applicable. However, at the same time new research questions and ideas for extensions of this work have become apparent.

Our method for the dynamic analysis of test specifications can be extended in various ways. First, the model can be refined to include more complex data types, such as arrays. This makes the semantics of the model more complicated, but allows a more compact formulation of certain properties. One example for such a property would be the violation of connection rules in TTCN-3. In general, the simpler the model is, the more verbose the phrasing of some properties on the model is. On the other hand, extending the model definition and its semantics has negative influence on the complexity of the reverse engineering algorithm. Therefore, the actual model definition is always a tradeoff between the complexity and the expressiveness or compactness of the model.

The second possible refinement concerns the reverse engineering itself. It can be extended to record more information about data and data-flow. That way, we are able to reduce the amount of abstraction and the amount of false positives in the analysis results. As the results from the case study have indicated, false positives do not seem to be a big problem for the kinds of TTCN-3 test specifications that we have analyzed. But the situation may be different for other languages or test development methodologies.

The specification of property-based events that need to be logged during the test specification simulation is currently realized by enriching the tree traversal code in our implementation. This is not intuitive from a user's point of view, especially, if the person is not familiar with the reverse engineering method or code. Therefore, we suggest the design and creation of a dedicated *Domain Specific Language* (DSL) that describes declaratively how and where exactly the test specification should produce property events, allowing the user to use only this DSL to produce different property events for analyzing different properties. In fact, such a DSL could be designed to be a generic instrumentation DSL for tree traversals which could be applicable to other uses as well.

An evolutionary analysis could provide more information about certain aspects of the test case, for example, how stable the test case is in its current state. For that purpose, the history of test cases derived from version control systems can be incorporated in the analysis interpretation. Similarly, a correlation between data of bug databases, the test specification code, and the version history would be very interesting. We expect that this additional data

can be very valuable for assessments, especially regarding qualitative statements about the current state of a test case and its future.

Finally, there certainly are other properties that can be analyzed using our methodology. It is likely that these are derived from actual practical needs and demands. Therefore, our property catalog is of introductory nature and we expect it to grow as experience shows what the actual demands are.

The response consistency criterion that we presented works well for the case that the SUT behaves deterministically. If this is not the case, we are confronted with the situation that responses are not necessarily the direct result of the stimuli, but also other factors may influence the responses. Therefore, the approach only to regard the stimuli may not be always enough in this case and the relation has to be refined.

Further possible improvements of the response inconsistency analysis are optimizations for the analysis of the test suite as a whole: the relation relates two test cases, but if we need to analyze a test suite completely for such inconsistencies, we need to compare every test case in the test suite to all the others. This is computationally expensive and can be improved, for example, by predetermining possible candidates by only partial analyses that take place before the actual comparison is started. We expect that the amount of comparisons can be reduced drastically using such methods, making the overall approach more feasible.

Response inconsistencies are only one specific kind of possible anomalies that test suites may exhibit when we analyze test cases to all the others. There are various other possible test suite anomalies that could extend the work of this thesis, such as *verdict inconsistency*. Verdict inconsistency implies that test cases set different test verdicts despite the fact that their observable traces match. Here, we assume that the same observable traces that are produced from the same initial state in the SUT must also lead to the same test verdicts.

Another kind of analysis is the identification of *test case behavior duplications*. Here, we could attempt to find behavior duplication either between two test cases as a whole in order to remove redundant test cases, or behavior parts, to find candidates for the application of *extract* refactorings. Such analyses may also regard different behavioral orderings that do not change the semantics of the test cases in order to identify test cases that do the same things.

Just as for the test case analysis, a further step would be to incorporate the history of different test specification versions into the analysis to derive trends from the past and to predict trends about the future of the test suite.

The work of this thesis followed the vision of a more rigorous and systematic analytical quality assurance for test specifications. Unfortunately, the systematic quality assurance of test specifications is still in the fledgling stages, even though test specifications grow rapidly in size and the importance of more robust and dependable systems is growing. We observe that the awareness about test specification quality and the need for active quality control of active test specification quality assurance in currently changing. Test specification development teams at ETSI, for example, actively seek tools that support them in the checking

of coding guidelines. We are confident that this kind of awareness is growing and hope that the results of this thesis provide a solid foundation for a more rigorous approach to the systematic automated analytical quality assurance in the everyday life of a test developer.

Bibliography

[1] Checkstyle 5.1. `http://checkstyle.sourceforge.net`. [04/19/2010].

[2] Java Decompiler. `http://java.decompiler.free.fr`. [04/19/2010].

[3] NuSMV 2.4.3. `http://nusmv.fbk.eu`. [04/19/2010].

[4] Santos Laoratory Specification Patterns. `http://patterns.projects.cis.ksu.edu`. [04/19/2010].

[5] J. Alilovic-Curgus and S. T. Vuong. A Metric Based Theory of Test Selection and Coverage. In *Proceedings of the IFIP TC6/WG6.1 Thirteenth International Symposium on Protocol Specification, Testing and Verification XIII*, pages 289–304. North-Holland, 1993.

[6] Apache Software Foundation. ActiveMQ 5.3.1. `http://activemq.apache.org`. [04/19/2010].

[7] N. Ayewah, D. Hovemeyer, J. D. Morgenthaler, J. Penix, and W. Pugh. Using Static Analysis to Find Bugs. *IEEE Software*, 25(5):22–29, 2008.

[8] V. R. Basili and D. M. Weiss. A Methodology for Collecting Valid Software Engineering Data. *IEEE Transactions on Software Engineering*, SE-10(6):728–738, 1984.

[9] G. Bath and J. McKay. *The Software Test Engineer's Handbook*. Rocky Nook, 2008.

[10] A. Bauer, M. Leucker, and C. Schallhart. Model-Based Runtime Analysis of Distributed Reactive Systems. In *Proceedings of the 2006 Australian Software Engineering Conference (ASWEC)*, Los Alamitos, CA, 2006. IEEE.

[11] K. Beck. *Extreme Programming Explained*. Addison Wesley, 2000.

[12] D. Bedrin. jtracert 0.1.3. `http://jtracert.googlecode.com`. [04/19/2010].

[13] B. Beizer. *Software Testing Techniques (2nd Edition)*. Van Nostrand Reinhold Co., 1990.

[14] M. Ben-Ari. *Principles of Concurrent and Distributed Programming: Algorithms and Models*. Prentice Hall, 2006.

[15] M. Ben-Ari. *Principle of the SPIN Model Checker*. Springer, 2008.

[16] B. Berard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, P. Schnoebelen, and P. McKenzie. *Systems and Software Verification: Model-Checking Techniques and Tools*. Springer, 2001.

[17] J.A. Bergstra and J.W. Klop. Algebra of Communicating Processes with Abstraction. *Theoretical Computer Science*, 37(1):77–121, 1985.

[18] M. Bisanz. Pattern-based Smell Detection in TTCN-3 Test Suites. Master's thesis, Institute for Informatics, University of Göttingen, Germany, ZFI-BM-2006-44, ISSN 1612-6793, 2007.

[19] R. Black. *Critical Testing Processes: Plan, Prepare, Perform, Perfect*. Addison Wesley, 2003.

[20] R. Black. *Advanced Software Testing Vol. 1: Guide to the ISTQB Advanced Certification as an Advanced Test Analyst*. Rocky Nook, 2008.

[21] R. Black. *Advanced Software Testing Vol. 2: Guide to the ISTQB Advanced Certification as an Advanced Test Manager*. Rocky Nook, 2009.

[22] B. Boehm. R & D Trends and Defense Needs. In P. Wegner, editor, *Research Directions in Software Technology*. MIT Press, 1979.

[23] B.W. Boehm, J.R. Brown, J.R. Kaspar, M. Lipow, C.J. MacLead, and M.J. Merrit. *Characteristics of Software Quality*. North Holland, 1978.

[24] S. Boroday, A. Petrenko, and A. Ulrich. Test Suite Consistency Verification. In *Proceedings of the 6th IEEE East-West Design & Test Symposium (EWDTS 2008), Ukraine*, 2008.

[25] L. C. Briand, Y. Labiche, and J. Leduc. Toward the Reverse Engineering of UML Sequence Diagrams for Distributed Java Software. *IEEE Transactions on Software Engineering*, 32(9):642–663, 2006.

[26] Carnegie Mellon Software Engineering Institute (SEI). *CMMI for Development (CMMI-DEV), Version 1.2*, August 2006.

[27] Carnegie Mellon Software Engineering Institute (SEI). *CMMI for Acquisition (CMMI-ACQ), Version 1.2*, November 2007.

[28] Carnegie Mellon Software Engineering Institute (SEI). *CMMI for Services (CMMI-SVC), Version 1.2*, February 2009.

[29] E. G. Cartaxo, F. G. O. Neto, and P. D. L. Machado. Automated Test Case Selection Based on a Similarity Function. In *Proceedings of the 2nd Workshop on Model-Based Testing (MOTES 2007)*, 2007.

[30] S. Chatterjee and A. S. Hadi. *Regression Analysis by Example*. Wiley, 2006.

[31] F. Chen and G. Roşu. MOP: An Efficient and Generic Runtime Verification Framework. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA 07)*. ACM, 2007.

[32] S. R. Chidamber and C. Kemerer. A Metric Suite for Object-Oriented Design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.

[33] E. J. Chikofsky and J. H. Cross II. Reverse Engineering and Design Recovery: A Taxonomy. *IEEE Software*, 7(1):13–17, January 1990.

[34] E. M. Clarke and E. A. Emerson. Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic. In *Logic of Programs, Workshop*, volume 131 of *Lecture Notes In Computer Science*. Springer, 1981.

[35] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(2):244–263, 1986.

[36] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 2000.

[37] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Pasareanu, Robby, and H. Zheng. Bandera: Extracting Finite-State Models From Java Source Code. In *International Conference on Software Engineering*. ACM, 2000.

[38] R. D. Craig and S. P. Jaskiel. *Systematic Software Testing*. Artech House, 2002.

[39] T. DeMarco. *Controlling Software Projects: Management, Measurement and Estimation*. Prentice Hall, 1982.

[40] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on Test Data Selection: Help for the Practicing Programmer. *IEEE Computer*, 11(4):34–41, April 1978.

[41] E. W. Dijkstra. Hierarchical Ordering of Sequential Processes. *Acta Informatica l*, 1(2):115–138, 1971.

[42] G. Din. *A Performance Test Design Method and its Implementation Patterns for Multi-Services Systems*. PhD thesis, Technical University Berlin, 2008.

[43] G. Din, D. Vega, and I. Schieferdecker. Automated Maintainability of TTCN-3 Test Suites Based on Guideline Checking. In *Software Technologies for Embedded and Ubiquitous Systems*, volume 5287 of *Lecture Notes in Computer Science*. Springer, 2008.

[44] R.G. Dromey. A Model for Software Product Quality. *IEEE Transactions on Software Engineering*, 21(2):146–162, 1995.

[45] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Property Specification Patterns for Finite-State Verification. In *Proceeding of the 2nd Workshop on Formal Methods in Software Practice (FMSP)*. ACM, 1998.

[46] Eclipse Foundation. Eclipse 3.5.2. `http://www.eclipse.org`. [04/19/2010].

[47] E. A. Emerson and E. M. Clarke. Characterizing Correctness Properties of Parallel Programs Using Fixpoints. In *Proceedings of the 7th Colloquium on Automata, Languages and Programming*, volume 85 of *Lecture Notes In Computer Science*. Springer, 1980.

[48] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon System for Dynamic Detection of Likely Invariants. *Science of Computer Programming*, 69(1–3):35–45, 2007.

[49] European Telecommunications Standards Institute (ETSI). *Technial Specification (TS) 102 385-3 V2.2.1 (2006-04): Conformance Testing for WiMAX/HiperMAN 1.2.1; Part 3: Abstract Test Suite (ATS)*, 2006.

[50] European Telecommunications Standards Institute (ETSI). *Technical Specification (TS) 102 027-3 V4.1.1 (2006-07): SIP ATS & PIXIT; Part 3: Abstract Test Suite (ATS) and partial Protocol Implementation eXtra Information for Testing (PIXIT)*, 2006.

[51] European Telecommunications Standards Institute (ETSI). *Technical Specification (TS) 102 516 V1.1 (2006-04): IPv6 Core Protocol; Conformance Abstract Test Suite (ATS) and partial Protocol Implementation eXtra Information for Testing (PIXIT)*, 2006.

[52] European Telecommunications Standards Institute (ETSI). *ETSI Technial Specification (ETS) 300 406 ed. 1: Methods for Testing and Specification (MTS); Protocol and Profile Conformance Testing Specifications; Standardization Methodology*, 1995.

[53] European Telecommunications Standards Institute (ETSI). *ETSI TS 102 210: Broadband Radio Access Networks (BRAN); HiperMAN; System Profiles*, 2005.

[54] European Telecommunications Standards Institute (ETSI). *ETSI TS 102 178 (v1.4.1): Broadband Radio Access Networks (BRAN); HiperMAN; Data Link Control (DLC) Layer*, 2007.

[55] European Telecommunications Standards Institute (ETSI). *ETSI ES 201 873 V3.4.1 (2008-09): The Testing and Test Control Notation version 3; Parts 1–10*, 2008.

[56] European Telecommunications Standards Institute (ETSI). *ETSI ES 201 873-4 V4.1.1 (2009-06); The Testing and Test Control Notation version 3; Part 4: TTCN-3 Operational Semantics*, 2009.

[57] European Telecommunications Standards Institute (ETSI). *ETSI ES 201 873-5 V4.1.1 (2009-06): The Testing and Test Control Notation version 3; Part 5: TTCN-3 Runtime Interface (TRI)*, 2009.

[58] European Telecommunications Standards Institute (ETSI). *ETSI ES 201 873-6 V4.1.1 (2009-06): The Testing and Test Control Notation version 3; Part 6: TTCN-3 Control Interface (TCI)*, 2009.

[59] C.-F. Fan and S. Yih. Prescriptive Metrics for Software Quality Assurance. In *Proceedings of the First Asia-Pacific Software Engineering Conference*. IEEE, 1994.

[60] Federal Ministry of Defense (BMVg), Federal Office of the Bundeswehr for Information Management and Information Technology (IT-AmtBw), and Federal Ministry of the Interior, Central Office for Information Technology Coordination in the Federal Administration (BMI-KBSt). *V-Modell 97*, 1997.

[61] N. E. Fenton and S. L. Pfleeger. *Software Metrics*. PWS Publishing Company, 1997.

[62] C. Flanagan and S. N. Freund. Dynamic Architecture Extraction. In *Formal Approaches to Software Testing and Runtime Verification*, volume 4262 of *Lecture Notes in Computer Science (LNCS)*. Springer, 2006.

[63] L. D. Fosdick and L. J. Osterweil. Data Flow Analysis in Software Reliability. *ACM Computing Surveys*, 8(3):305–330, 1976.

[64] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[65] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.

[66] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems – An Approach to the State-Explosion Problem*, volume 1032 of *Lecture Notes in Computer Science (LNCS)*. Springer, 1996.

[67] J. Grabowski, D. Hogrefe, G. Réthy, I. Schieferdecker, A. Wiles, and C. Willcock. An Introduction into the Testing and Test Control Notation (TTCN-3). *Computer Networks*, 42(3):375–403, 2003.

[68] R. Grady. *Practical Software Metrics for Project Management and Process Improvement*. Prentice Hall, 1992.

[69] H. Hallal, S. Boroday, A. Ulrich, and A. Petrenko. An Automata-Based Approach to Property Testing in Event Traces. In D. Hogrefe and A. Wiles, editors, *Testing of Communicating Systems*, volume 2644 of *Lecture Notes in Computer Science (LNCS)*. Springer, 2003.

[70] H. H. Hallal, S. Boroday, A. Petrenko, and A. Ulrich. A Formal Approach to Property Testing in Causally Consistent Distributed Traces. *Formal Aspects of Computing*, 18(1), 2006.

[71] M. H. Halstead. *Elements of Software Science*. Elsevier, 1977.

[72] A. Hamou-Lhadj, E. Braun, D. Amyot, and T. Lethbridge. Recovering Behavioral Design Models from Execution Traces. In *Proceedings of the Ninth European Conference on Software Maintenance and Reengineering (CSMR 2005)*. IEEE, 2005.

[73] K. Havelund. Java PathFinder, A Translator from Java to Promela. In *Theoretical and Practical Aspects of SPIN Model Checking: Proceedings of the 5th and 6th International SPIN Workshops*, volume 1680 of *Lecture Notes in Computer Science (LNCS)*. Springer, 1999.

[74] K. Havelund. Runtime Verification of C Programs. In *Testing of Software and Communicating Systems, 20th IFIP TC 6/WG 6.1 International Conference, TestCom 2008, 8th International Workshop, FATES 2008*, volume 5047 of *Lecture Notes in Computer Science (LNCS)*. Springer, 2008.

[75] S. Henry, D. Kafura, and K. Harris. On the Relationships Among Three Software Metrics. In *Proceedings of the 1981 ACM Workshop/Symposium on Measurement and Evaluation of Software Quality*. ACM, 1981.

[76] R. Hinden and S. Deering. *RFC 3513: Internet Protocol Version 6 (IPv6) Addressing Architecture*. IETF, 2003.

[77] C. A. R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, 1978.

[78] D. W. Hoffmann. *Software-Qualität*. Springer, 2008.

[79] G. J. Holzmann. *The Spin Model Checker. Primer and Reference Manual.* Addison Wesley Longman, 2003.

[80] G. J. Holzmann and M. H. Smith. Software Model Checking: Extracting Verification Models From Source Code. *Software Testing, Verification and Reliability*, 11(2):65–79, 2001.

[81] G. J. Holzmann and M. H. Smith. An Automated Verification Method for Distributed Systems Software Based on Model Extraction. *IEEE Transactions on Software Engineering*, 28(4):364–377, 2002.

[82] IBM. Rational Purify 7.0.1. `http://www.ibm.com/software/awdtools/purify`. [04/19/2010].

[83] Institute of Electrical and Electronics Engineers (IEEE). *IEEE Standard 802.16-2004: IEEE Standard for Local and Metropolitan Area Networks - Part 16: Air Interface for Fixed Broadband Wireless Access Systems*, 2004.

[84] Institute of Electrical and Electronics Engineers (IEEE). *IEEE 802.16e-2005: IEEE Standard for Local and Metropolitan Area Networks - Part 16: Air Interface for Fixed and Mobile Broadband Wireless Access Systems - Amendment for Physical and Medium Access Control Layers for Combined Fixed and Mobile Operation in Licensed Bands*, 2005.

[85] ISO/IEC. International standard ISO/IEC 9646-3:1998: Information Technology – Open Systems Interconnection – Conformance testing methodology and framework – Part 3: The Tree and Tabular Combined Notation (TTCN). International Organization for Standardization/International Electrotechnical Commission.

[86] ISO/IEC. Information Technology – Open Systems Interconnection – Conformance testing methodology and framework. International ISO/IEC multipart standard No. 9646, 1994-1997.

[87] ISO/IEC. International Standard No. 14598: Information technology — Software product evaluation; Parts 1–6. International Organization for Standardization (ISO) / International Electrotechnical Commission (IEC), Geneva, Switzerland, 1998-2001.

[88] ISO/IEC. International Standard No. 9126: Software engineering – Product quality; Parts 1–4. International Organization for Standardization (ISO) / International Electrotechnical Commission (IEC), Geneva, Switzerland, 2001-2004.

[89] ISO/IEC. International Standard No. 15504: Information technology – process assessment; Parts 1–7. International ISO/IEC multipart standard No. 15504, 2003-2008.

[90] ISO/IEC. International Standard No. 25000: Software Engineering – Software product Quality Requirements and Evaluation (SQuaRE). International Organization for

Standardization (ISO) / International Electrotechnical Commission (IEC), Geneva, Switzerland, 2005.

[91] W. Thomson (Lord Kelvin). Popular Lectures and Addresses. 1889-1894.

[92] M. G. Kendall. A New Measure of Rank Correlation. *Biometrika*, 30(1/2):81–93, 1938.

[93] T. Koomen and M. Pol. *Test Process Improvement: A Practical Step-By-Step Guide to Structured Testing*. Addison Wesley, 1999.

[94] P. Kruchten. *The Rational Unified Process. An Introduction.* Addison Wesley, 2003.

[95] N.A. Lynch and M.R. Tuttle. Hierarchical Correctness Proofs for Distributed Algorithms. In *Proceedings of the 6th ACM Symposium on Principles of Distributed Computing (PODC)*. ACM, 1987.

[96] T. J. McCabe. A Complexity Measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, 1976.

[97] J. A. McCall, P. K. Richards, and G. F. Walters. Factors in Software Quality. Technical Report RADC TR-77-369, US Rome Air Development Center, 1977.

[98] J. McCann, S. Deering, and J. Mogul. *RFC 1981: Path MTU Discovery for IP version 6*. IETF, 1996.

[99] K. L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. PhD thesis, Carnegie Mellon University, 1992.

[100] D. Megginson. Simple API for XML. `http://www.saxproject.org`. [04/19/2010].

[101] P. Mehlitz, D. Giannakopoulou, C. Pasareanu, and M. Mansouri-Samani. Java PathFinder (JPF). `http://babelfish.arc.nasa.gov/trac/jpf`. [04/19/2010].

[102] E. Merks. Eclipse Modeling Framework (EMF) 2.5. `http://www.eclipse.org/emf`. [04/19/2010].

[103] G. Meszaros. *xUnit Test Patterns: Refactoring Test Code*. Addison-Wesley, 2007.

[104] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, 1980.

[105] G. Myers. *The Art of Software Testing*. Wiley, 1979.

[106] D. Neumann. Entwurf und Weiterverarbeitung eines XML-Formats zur Speicherung von Transitionssystemen (Projektbericht). Technical report, Zentrum für Informatik, Georg August University Göttingen, 2008.

[107] Nomagic. Magicdraw 16.6. `http://www.magicdraw.com`. [04/19/2010].

[108] Institute of Electrical and Electronics Engineers (IEEE). *IEEE Standard 610.12: IEEE standard glossary of software engineering terminology*, 1990.

[109] Institute of Electrical and Electronics Engineers (IEEE). *IEEE Standard 1061-1998: IEEE Standard for a Software Quality Metrics Methodology*, 1998.

[110] OMG. UML Testing Profile (Version 1.0 formal/05-07-07). Object Management Group (OMG), 2007.

[111] Object Management Group (OMG). Unified Modeling Language (UML) 2.2 Infrastructure Specification (formal/2009-02-04), 2009.

[112] Object Management Group (OMG). Unified Modeling Language (UML) 2.2 Superstructure Specification (formal/2009-02-02), 2009.

[113] Oracle. Java Architecture for XML Binding (JAXB). `https://jaxb.dev.java.net`. [04/19/2010].

[114] Oracle. Java Message Service Specifiation - Version 1.1. `https://java.sun.com/products/jms`. [04/19/2010].

[115] Oracle. Javadoc. `http://java.sun.com/j2se/javadoc`. [04/19/2010].

[116] D. L. Parnas. Software Aging. In *Proceedings of the 16th International Conference on Software Engineering (ICSE), May 16–21, 1994, Sorrento, Italy*, pages 279–287. IEEE/ACM, 1994.

[117] T. Parr. ANTLR parser generator 3.2. `http://www.antlr.org`. [04/19/2010].

[118] K. Pearson. On Lines and Planes of Closest Fit to Systems of Points in Space. *Philosophical Magazine*, 2(6):559–572, 1901.

[119] G. Plotkin. A Structural Approach to Operational Semantics. Technical report, Department of Computer Science, Aarhus University, Denmark, 1981.

[120] PMD 4.2.5. `http://pmd.sourceforge.net`. [04/19/2010].

[121] A. Pnueli. The Temporal Logic of Programs. In *Proceedings of the 18th Annual IEEE Symposium on the Foundations of Computer Science (FOCS)*. IEEE, 1977.

[122] J. P. Queille and J. Sifakis. Specification and Verification of Concurrent Systems in CESAR. In *Proceedings of the 5th Colloquium on International Symposium on Programming*. Springer, 1982.

[123] AT & T Research. Graphviz - Graph Visualization Software 2.26. `http://www.graphviz.org`. [04/19/2010].

[124] H. G. Rice. Classes of Recursively Enumerable Sets and Their Decision Problems. *Transactions of the American Mathematical Society*, 74(2):358–366, 1953.

[125] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler. *RFC 3261: SIP: Session Initiation Protocol*. IETF, 2002.

[126] D. S. Rosenblum. Formal Methods and Testing: Why the State-of-the-Art is not the State-of-the-Practice. *ACM SIGSOFT Software Engineering Notes*, 21(4):64–66, 1996.

[127] Hex-Rays SA. The IDA Pro Disassembler and Debugger 5.0. `http://www.hex-rays.com/idapro`. [04/19/2010].

[128] I. Schieferdecker and G. Din. A Meta-Model for TTCN-3. In M. Núñez, Z. Maamar, F.L. Pelayo, K. Pousttchi, and F. Rubio, editors, *Applying Formal Methods: Testing, Performance and M/ECommerce, FORTE 2004 Workshops, Toledo, Spain, October 1–2, 2004*, volume 3236 of *Lecture Notes in Computer Science (LNCS)*. Springer, 2004.

[129] K. Schneider. *Verification of Reactive Systems*. Springer, 2003.

[130] D. Schuler and A. Zeller. Javalanche: Efficient Mutation Testing for Java. In *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*. ACM, 2009.

[131] J. Seward, N. Nethercote, and J. Weidendorfer. *Valgrind 3.3 - Advanced Debugging and Profiling for Gnu/Linux Applications*. Network Theory Ltd, 2008.

[132] H. M. Sneed, M. Baumgartner, and R. Seidl. *Der Systemtest – Von den Anforderungen zum Qualitätsnachweis*. Carl Hanser, 2009.

[133] C. Spearman. The Proof and Measurement of Association Between Two Things. *The American Journal of Psychology*, 15(1):72–101, 1904.

[134] A. Spillner, T. Linz, and H. Schaefer. *Software Testing Foundations: A Study Guide for the Certified Tester Exam*. Rocky Nook, 2007.

[135] T. Tamai. Social Impact Of Information System Failures. *IEEE Computer*, 42(6):58–65, 2009.

[136] The Runtime Reflection Project Team. Runtime Reflection. `http://runtime.in.tum.de`. [04/19/2010].

[137] H. S. Thomson, D. Beech, M. Maloney, and N. Mendelsohn. *XML Schema Parts 0-2: [Primer, Structures, Datatypes] Second Edition*. W3C, 2004.

[138] TMMI Foundation. TMMi Reference Model Version 2.0, 2009.

[139] J. Tretmans. Test Generation with Inputs, Outputs, and Quiescence. In *Proceedings of Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, volume 1055 of *Lecture Notes in Computer Science (LNCS)*. Springer, 1996.

[140] A. M. Turing. On Computable Numbers, With an Application to the Enscheidungsproblem. In *Procedings of the London Mathematical Society*, volume 42, pages 230–265, 1936.

[141] M.R. Tuttle. Hierarchical Correctness Proofs for Distributed Algorithms. Master's thesis, Department of Electrical Engineering and Computer Science, MIT, United States, 1987.

[142] A. Ulrich and A. Petrenko. Reverse Engineering Models from Traces to Validate Distributed Systems - An Industrial Case Study. In *Proceedings of the European Conference on Model Driven Architecture: Foundations and Applications (ECMDA-FA 2007)*, volume 4530 of *Lecture Notes in Computer Science (LNCS)*. Springer, 2007.

[143] Uppsala Universitet and Aalborg University. Uppaal. `http://www.uppaal.com`. [04/19/2010].

[144] M. Utting and B. Legeard. *Practical Model-Based Testing – A Tools Approach*. Morgan Kaufmann, 2007.

[145] D. van Heesch. Doxygen. `http://www.doxygen.org`. [04/19/2010].

[146] C.J. van Rijsbergen. *Information Retrieval*. Butterworth-Heinemann, 1979.

[147] D. Vega, G. Din, S. Taranu, and I. Schieferdecker. Application of Clustering Methods for Analysing of TTCN-3 Test Data Quality. In *Proceedings of the 2008 The Third International Conference on Software Engineering Advances (ICSEA 2008)*. IEEE, 2008.

[148] D. Vega and I. Schieferdecker. Towards Quality of TTCN-3 Tests. In *Proceedings of SAM'06: Fifth Workshop on System Analysis and Modelling*, volume 4320 of *Lecture Notes in Computer Science (LNCS)*. Springer, 2006.

[149] D. Vega, I. Schieferdecker, and G. Din. Test Data Variance as a Test Quality Measure: Exemplified for TTCN-3. In *Testing of Software and Communicating Systems*, volume 4581 of *Lecture Notes in Computer Science*. Springer, 2007.

[150] A. Vouffo-Feudjio and I. Schiefer. Test Patterns with TTCN-3. In *Formal Approaches to Softwar Testing*, pages 150–179. Springer, 2005.

[151] N. Walkinshaw, K. Bogdanov, and M. Holcombe. Identifying State Transitions and their Functions in Source Code. In *Proceedings of the Testing: Academic & Industrial Conference on Practice And Research Techniques (TAIC-PART 2006)*. IEEE, 2006.

[152] E. Werner, J. Grabowski, H. Neukirchen, N. Röttger, S. Waack, and B. Zeiss. TTCN-3 Quality Engineering: Using Learning Techniques to Evaluate Metric Sets. In *Proceedings of 13th System Design Language Forum (SDL Forum 2007)*, volume 4745 of *Lecture Notes in Computer Science (LNCS)*. Springer, 2007.

[153] H. Yuan and T. Xie. Automatic Extraction of Abstract-Object-State Machines Based on Branch Coverage. In *1st International Workshop on Reverse Engineering To Requirements (RETR 2005)*. IEEE, 2005.

[154] B. Zeiss. A Refactoring Tool for TTCN-3. Master's thesis, Institute for Informatics, University of Göttingen, Germany, ZFI-BM-2006-05, 2006.

[155] B. Zeiss and J. Grabowski. Analyzing Response Inconsistencies in Test Suites. In *Proceedings of the 21st IFIP International Conference on Testing of Communicating Systems and the 9th Int. Workshop on Formal Approaches to Testing of Software (TESTCOM/FATES 2009*, volume 5826 of *Lecture Notes in Computer Science (LNCS)*. Springer, 2009.

[156] B. Zeiss, D. Vega, I. Schieferdecker, H. Neukirchen, and J. Grabowski. Applying the ISO 9126 Quality Model to Test Specifications – Exemplified for TTCN-3 Test Specifications. In *Proceedings of Software Engineering 2007 (SE 2007)*, volume 105 of *Lecture Notes in Informatics (LNI)*. Köllen Verlag, 2007.

A. Appendix

The appendix complements the thesis with more detailed descriptions of algorithms and examples that we felt were too detailed, complicated, or formal for the main text. Nevertheless, the contents are important results of the thesis and form a considerable part of the overall contribution.

A.1. Model Reconstruction Algorithm

In the following, we present the model increment reconstruction algorithm. It takes as input a log λ_i with events $\rho_i, 0 < i < m$. The model increment reconstruction is applied to all $\lambda_0, \lambda_1, \dots, \lambda_n$ where all n logs represent executions and are expected to have a complete branch coverage of the test specification under analysis. The result after each increment is an EMIOTS model which is refined according to the new information that is provided in each log.

Algorithm A.1 (Model Increment Reconstruction) Each log λ_i representing a test execution may contain events of multiple test processes $1, \dots, l$ (due to the *pid* field in the event) that are each represented by the target EMIOTS models $M_j, 1 \leq j \leq l$.

1. Create subsets for each process $log^j := \{\rho_k | \rho_k^{pid} = j\}$ and sequences $logseq^j : \mathbb{N} \rightarrow log^j, i \mapsto logseq_i^j$ that are partitioned according to the process ids while retaining the order of their event occurrences.
2. Let $s_{current} \leftarrow \varepsilon$ and $s_{next} \leftarrow \varepsilon$ be uninitialized state variables.
3. Let $proctrans \subseteq S \times A \times S \times \mathbb{N} \times \mathbb{N}, proctrans = \emptyset$ be an initially empty set of tuples containing a transition, a scope id, and an event id. It will be used to identify which transition have been processed already.
4. Let $\zeta \leftarrow \varepsilon$ denote an uninitialized stack of natural numbers and ζ_{seq} denote the sequence of natural numbers that have been pushed on to the stack.
5. Let *sval* be an injetive function $sval : K \rightarrow \mathbb{N}$ that maps a sequence of integers K to one value. In practice, *sval* could be a hash function with as few collisions as possible.
6. For each $logseq_i^j, 0 < i < |logseq_i^j|$, repeat the following steps:

- Set $event \leftarrow logseq_i^j$ and $event_{next} \leftarrow logseq_{i+x}^j$. Where $x > 0$ represents the nearest possible event that is not a scope start event, i.e. $NEXT := \{logseq_{i+x}^j | etype(logseq_{i+x}^j) \neq \text{scopeStartEvent}, x > 0\}, x \in NEXT$ and $\forall y \in NEXT : x \leq y$.
- If $M_j^S = \emptyset$, create a start state s_0 and add it to M_j^S. Set $s_{current} \leftarrow s_0$.
- If $etype(event) = \text{scopeStartEvent}$, then push $id(event)$ on to the stack ζ.
- If $etype(event) = \text{scopeEndEvent}$, then
  - Pop the stack $zeta$.
  - If $\{s' | (s_{current}, \tau, s', sval(\zeta_{seq}), id(event)) \in proctrans\} \neq \emptyset$, then set $s_{current} \leftarrow s'$ and skip to the next iteration of 6.
  - If there is an action a' corresponding to $event_{next}$ and $\{a' | (s'', a', s''', sval(\zeta'_{seq}), id(event_{next})) \in proctrans\} \neq \emptyset$, set state $s_{next} \leftarrow s''$ where $\zeta'_{seq} = \zeta_{seq}$ if $etype(event_{next}) \neq \text{scopeEndEvent}$ and $\zeta'_{seq} = \zeta.pop()_{seq}$ if $etype(event_{next}) = \text{scopeEndEvent}$. Otherwise, create a new state s_{next}.
  - Add s_{next} to M_j^S, add $(s_{current}, d(event), s_{next})$ to M_j^λ. Add $(s_{current}, \tau, s_{next}, sval(\zeta_{seq}), id(event))$ to the $proctrans$.
- If $etype(event) = \text{messageInputEvent} \vee etype(event) = \text{messageOutputEvent}$, then
  - If $etype(event) = \text{messageInputEvent}$, then set $a \leftarrow (i(event))$, otherwise set $a \leftarrow (o(event))$.
  - If $\{s' | (s_{current}, a, s', sval(\zeta_{seq}), id(event)) \in proctrans\} \neq \emptyset$, then set $s_{current} \leftarrow s'$ and skip to the next iteration of 6.
  - If there is an action a' corresponding to $event_{next}$ and $\{a' | (s'', a', s''', sval(\zeta_{seq}), id(event_{next})) \in proctrans\} \neq \emptyset$, set state $s_{next} \leftarrow s''$. Otherwise, create a new state s_{next}.
  - Add s_{next} to M_j^S, add $(s_{current}, a, s_{next})$ to M_j^λ, and add a to M_{A_I} or M_{A_O} respectively. Add $(s_{current}, a, s_{next}, sval(\zeta_{seq}), id(event))$ to $proctrans$.
  - If $l(event) = 0$, set $s_{current} \leftarrow s_{next}$.
- If $etype(event) = \text{dataEvent}$, then
  - Data events are found in event sequences which are combined into a single transition. Therefore, we find the last data event $event_{lprop} \leftarrow logseq_k^j$ that directly follows $event$.
  - If there is is an action $a \in M_{A_U}$ and $\{s' | (s_{current}, a, s', sval(\zeta_{seq}), id(event)) \in proctrans\} \neq \emptyset$ where the variable update for v_i corresponds to $pn(logseq_l^j)$ and sets the value $pv(logseq_l^j), i \leq l \leq k$, then set $s_{current} \leftarrow s'$ and skip to the next iteration of 6 and skip the index i to $k+1$.
  - If there is an action a' corresponding to event $logseq_{k+1}^j$ and

$\{a'|(s'', a', s''', sval(\zeta_{seq}), id(event_{next})) \in proctrans\} \neq \emptyset$, set state $s_{next} \leftarrow s''$. Otherwise, create a new state s_{next}.

  - Add s_{next} to M_j^S, add $(s_{current}, a, s_{next})$ to M_j^λ, and add a to M_{A_U}. Add $(s_{current}, a, s_{next}, sval(\zeta_{seq}), id(event))$ to $proctrans$. Skip the index i to $k+1$.
  - If $l(event) = 0$, set $s_{current} \leftarrow s_{next}$.
- If $etype(event) =$ internalEvent, then
  - If there is is an action a in M_{A_N} that matches $d(event)$ and $\{s'|(s_{current}, a, s', sval(\zeta_{seq}), id(event)) \in proctrans\} \neq \emptyset$, then set $s_{current} \leftarrow s'$ and skip to the next iteration of 6.
  - If there is an action a' corresponding to $event_{next}$ and $\{a'|(s'', a', s''', sval(\zeta_{seq}), id(event_{next})) \in proctrans\} \neq \emptyset$, set state $s_{next} \leftarrow s''$. Otherwise, create a new state s_{next}.
  - Add s_{next} to M_j^S, add $(s_{current}, d(event), s_{next})$ to M_j^λ, and add a to M_A. Add $(s_{current}, d(event), s_{next}, sval(\zeta_{seq}), id(event))$ to the $proctrans$.
  - If $l(event) = 0$, set $s_{current} \leftarrow s_{next}$.

Intuitively, we convert the events into actions and insert states before and after the action. We identify equal transitions by the *proctrans* set which contains tuples with transitions, scope ids, and event ids. The stack *zeta* tracks scopes and by pushing and popping the current concrete event id. Thus, the whole stack in its order represents a scope history. This is important for the following reason: a code location, for example, identified by a line number is ambiguous in the context of program behavior. A code position can be executed from multiple other code locations, e.g., by procedure calls or jumps within the code. What makes a code location unique in its behavioral context is its scope history. Whenever a code location is executed in a different context, the scope history must always be different. By mapping each scope history to a unique value using *sval* and regarding both scope history and the event id, we can identify unique control-flow positions and reconstruct an interprocedural model. Identical states are identified by looking into the next event and its corresponding action. If it exists in the set *proctrans*, we need to connect the existing next state with the current state using the current event action. Finally, look-ahead events are also added to the incremental model, but ignored when stepping into a next state/event analysis.

A.2. Model Reconstruction Example

To illustrate the algorithm on a small example, we reconstruct the model based on one test component from the logs of Tables A.1 and A.2 step by step. The frame boxes denotes the current state of the variables of interest. We will omit the notation of M_1^λ as it is effectively as subset of *proctrans* with less information.

A.2.0.5. Preparations

- The events are partitioned into sequences $logseq^j$ according to the component on which the event took place. In our case, we deal with one component only and thus have a single sequenc $logseq^1$ that needs to be processed.
- $s_{current}$ and s_{next} are uninitialized, the set *proctrans* and stack ζ are empty.
- For the *sval* function, we choose a very simple solution for this example to keep the calculation easy: each value on the stack is assigned a prime factor and the stack value represents its respective multiplicity. For example, a if $\zeta_{seq} = (3,2,2)$, then $sval(\zeta_{seq}) = 2^3 + 3^2 + 5^2 = 42$. This ensures unique prime factorizations for each stack configuration by definition and thus also unique value mappings of ζ_{seq}. In practice, a better solution would be to use common hash functions that are nearly collision free.
- The set of variables is $(v_1, v_2, v_3, v_4) \in M_1^V$ where v_1 represents *v_pass*, v_2 represents *v_fail*, v_3 represents *v_inconc*, and v_4 represents *v_none*.

A.2.0.6. Example Log 1

- M_j^S is empty, so we create a start state s_0 with $s_{current} \leftarrow s_0$.

$$
\begin{aligned}
&\zeta_{seq} = \{\},\\
&proctrans = \{\},\\
&sval(\zeta_{seq}) = 0,\\
&s_{current} = s_0,\\
&M_1^S = \{s_0\},\\
&M_1^A = \{\}
\end{aligned}
$$

- The first event is a scope start event so we push 1 on to ζ.

$$
\begin{aligned}
&\zeta_{seq} = \{1\},\\
&proctrans = \{\},\\
&sval(\zeta_{seq}) = 2^1,\\
&s_{current} = s_0,
\end{aligned}
$$

event	type	pid	id	ss	se	l	i	o	d	pn	pv
1	ss	1	1	1	-	-	-	-	-	-	-
2	pe	1	2	-	-	-	-	-	-	v_pass	0
3	pe	1	3	-	-	-	-	-	-	v_fail	0
4	pe	1	4	-	-	-	-	-	-	v_inconc	0
5	pe	1	5	-	-	-	-	-	-	v_none	1
6	moe	1	6	-	-	-	-	!p1.m1	-	-	-
7	mie	1	7	-	-	1	?p1.m2	-	-	-	-
8	mie	1	8	-	-	1	?p1.*	-	-	-	-
9	mie	1	7	-	-	-	?p1.m2	-	-	-	-
10	ss	1	9	1	-	-	-	-	-	-	-
11	pe	1	10	-	-	-	-	-	-	v_pass	1
12	pe	1	11	-	-	-	-	-	-	v_fail	0
13	pe	1	12	-	-	-	-	-	-	v_inconc	0
14	pe	1	13	-	-	-	-	-	-	v_none	0
15	se	1	14	-	1	-	-	-	-	-	-
16	ss	1	15	1	-	-	-	-	-	-	-
17	moe	1	16	-	-	-	-	!p2.m3	-	-	-
18	mie	1	17	-	-	1	?p1.m4	-	-	-	-
19	mie	1	18	-	-	1	?p1.*	-	-	-	-
20	mie	1	17	-	-	-	?p1.m4	-	-	-	-
21	ss	1	19	1	-	-	-	-	-	-	-
22	pe	1	20	-	-	-	-	-	-	v_pass	1
23	pe	1	21	-	-	-	-	-	-	v_fail	0
24	pe	1	22	-	-	-	-	-	-	v_inconc	0
25	pe	1	23	-	-	-	-	-	-	v_none	0
26	se	1	24	-	1	-	-	-	-	-	-
27	se	1	25	-	1	-	-	-	-	-	-
28	se	1	26	-	1	-	-	-	-	-	-

Table A.1.: Example Log 1

$M_1^S = \{s_0\}$,
$M_1^A = \{\}$

- Events 2 to 5 are a sequence of proposition events. We create a new state s_1 with $s_{next} \leftarrow s_1$, an action $a_0 \in A_U$ reflecting the variable updates $(v_1, v_2, v_3, v_4) \rightarrow (0,0,0,1)$, a transition (s_0, a, s_1) adds them to their respective model sets. We update *proctrans* by adding $(s_0, a_0, s_1, 2^1, 2)$ and move forward by setting $s_{current} \leftarrow s_1$.

$\zeta_{seq} = \{1\}$,
$proctrans = \{((s_0, a_0, s_1, 2, 2))\}$,
$sval(\zeta_{seq}) = 2^1$,
$s_{current} = s_1$,

$M_1^S = \{s_0, s_1\},$
$M_1^A = \{a_0\}$

- Event 6: Create $a_1 \leftarrow !p1.m1 \in A_O$, create a new state s_2 with $s_{next} \leftarrow s_2$, a transition $(s1, a_1, s2)$ and add them to their respective model sets. We update *proctrans* by adding $(s_1, a_1, s_2, 2, 6)$ and move forward by setting $s_{current} \leftarrow s_2$.

$\zeta_{seq} = \{1\},$
$proctrans = \{(s_0, a_0, s_1, 2, 2), (s_1, a_1, s_2, 2, 6)\},$
$sval(\zeta_{seq}) = 2^1,$
$s_{current} = s_2,$
$M_1^S = \{s_0, s_1, s_2\},$
$M_1^A = \{a_0, a_1\}$

- Events 7 and 8 are message input look-ahead events. This means, they are processed like the other events, but $s_current$ is not altered.

$\zeta_{seq} = \{1\},$
$proctrans = \{(s_0, a_0, s_1, 2^1, 2), (s_1, a_1, s_2, 2^1, 6), (s_2, a_2, s_3, 2^1, 7), (s_2, a_3, s_4, 2^1, 8)\},$
$sval(\zeta_{seq}) = 2^1,$
$s_{current} = s_2,$
$M_1^S = \{s_0, s_1, s_2, s_3, s_4\},$
$M_1^A = \{a_0, a_1, a_2, a_3\}$

- Event 9 is a concrete message input event which has been announced by a look-ahead event already. As a result, we find out that there is an s_3 for which an element $(s_2, a_2, s_3, 2^1, 7) \in proctrans$ exists. Thus, we set $s_{current} \leftarrow s_3$ and proceed to the next event without any updates.
- Events 10–28 are handled in a similar way as the previous ones without any special cases as this is only the first iteration. The only noteworthy ones are events 15, 26, 27, 28 where τ transitions are inserted in order to model join points correctly within the control-flow. We only provide the end state for the remainder.

$\zeta_{seq} = \{\},$
$proctrans = \{(s_0, a_0, s_1, 2^1, 2), (s_1, a_1, s_2, 2^1, 6), (s_2, a_2, s_3, 2^1, 7), (s_2, a_3, s_4, 2^1, 8),$
$(s_3, a_4, s_5, 2^1 + 3^9, 10), (s_5, \tau, s_6, 2^1 + 3^9, 14), (s_6, a_6, s_7, 2^1 + 3^{15}, 16),$
$(s_7, a_7, s_8, 2^1 + 3^{15}, 17), (s_7, a_8, s_8, 2^1 + 3^{15}, 18), (s_8, a_9, s_{10}, 2^1 + 3^{15} + 5^{19}, 20),$
$(s_{10}, \tau, s_{11}, 2^1 + 3^{15} + 5^{19}, 24), (s_{11}, a_{11}, s_{12}, 2^1 + 3^{15}, 25), (s_{12}, \tau, s_{13}, 2^1, 26)\},$
$sval(\zeta_{seq}) = 0,$

event	type	pid	id	ss	se	l	i	o	d	pn	pv
1	ss	1	1	1	-	-	-	-	-	-	-
2	pe	1	2	-	-	-	-	-	-	v_pass	0
3	pe	1	3	-	-	-	-	-	-	v_fail	0
4	pe	1	4	-	-	-	-	-	-	v_inconc	0
5	pe	1	5	-	-	-	-	-	-	v_none	1
6	moe	1	6	-	-	-	-	!p1.m1	-	-	-
7	mie	1	7	-	-	1	?p1.m2	-	-	-	-
8	mie	1	8	-	-	1	?p1.*	-	-	-	-
9	mie	1	8	-	-	-	?p1.*	-	-	-	-
10	ss	1	27	1	-	-	-	-	-	-	-
11	pe	1	28	-	-	-	-	-	-	v_pass	0
12	pe	1	29	-	-	-	-	-	-	v_fail	1
13	pe	1	30	-	-	-	-	-	-	v_inconc	0
14	pe	1	31	-	-	-	-	-	-	v_none	0
15	se	1	32	-	1	-	-	-	-	-	-
16	ss	1	15	1	-	-	-	-	-	-	-
17	moe	1	16	-	-	-	-	!p2.m3	-	-	-
18	mie	1	17	-	-	1	?p1.m4	-	-	-	-
19	mie	1	18	-	-	1	?p1.*	-	-	-	-
20	mie	1	18	-	-	-	?p1.*	-	-	-	-
21	ss	1	33	1	-	-	-	-	-	-	-
22	pe	1	34	-	-	-	-	-	-	v_pass	0
23	pe	1	35	-	-	-	-	-	-	v_fail	1
24	pe	1	36	-	-	-	-	-	-	v_inconc	0
25	pe	1	37	-	-	-	-	-	-	v_none	0
26	se	1	38	-	1	-	-	-	-	-	-
27	se	1	25	-	1	-	-	-	-	-	-
28	se	1	26	-	1	-	-	-	-	-	-

Table A.2.: Example Log 2

$s_{current} = s_13$,
$M_1^S = \{s_0, s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8, s_9, s_{10}, s_{11}, s_{12}, s_{13}\}$,
$M_1^A = \{a_0, a_1, a_2, a_3, a_4, a_6, a_7, a_8, a_9, a_{11}, a_{12}\}$

A.2.0.7. Example Log 2

- In events 1–8, we detect that all potential transitions are already part of the model and the *proctrans* set. Thus, up to event 8, we essentially only “track” the log within our model such that $s_{current} = s_2$.
- Event 9 is the first new message input event in the second log, but the corresponding state s_4 and transition (s_2, a_3, s_4) have been added already. Thus, only $s_{current}$ moves forward to s_4.

- Event 10 introduces a new scope, but no new model elements so that $\zeta_{seq} = \{27, 1\}$ and $sval(\zeta_{seq}) = 2^1 + 3^27$.
- Event 11–14 is a sequence of proposition events that have not been added to the model already within this new scope.

$$\zeta_{seq} = \{27,1\},$$
$$proctrans = \{(s_0,a_0,s_1,2^1,2),(s_1,a_1,s_2,2^1,6),(s_2,a_2,s_3,2^1,7),(s_2,a_3,s_4,2^1,8),$$
$$(s_3,a_4,s_5,2^1+3^9,10),(s_5,\tau,s_6,2^1+3^9,14),(s_6,a_6,s_7,2^1+3^{15},16),$$
$$(s_7,a_7,s_8,2^1+3^{15},17),(s_7,a_8,s_8,2^1+3^{15},18),(s_8,a_9,s_{10},2^1+3^{15}+5^{19},20),$$
$$(s_{10},\tau,s_{11},2^1+3^{15}+5^{19},24),(s_{11},a_{11},s_{12},2^1+3^{15},25),(s_{12},\tau,s_{13},2^1,26),$$
$$(s_4,a_{13},s_{14},2^1+3^27,28)\},$$
$$sval(\zeta_{seq}) = 2^1+3^27,$$
$$s_{current} = s_14,$$
$$M_1^S = \{s_0,s_1,s_2,s_3,s_4,s_5,s_6,s_7,s_8,s_9,s_{10},s_{11},s_{12},s_{13},s_{14}\},$$
$$M_1^A = \{a_0,a_1,a_2,a_3,a_4,a_6,a_7,a_8,a_9,a_{11},a_{12},a_{13}\}$$

- Event 15 is the first ocurring merge event. A look-ahead to the next event respecting the future stack state identifies that $(s_6, a_6, s_7, 2^1 + 3^{15}, 16)$ already exists in *proctrans*. As a result, the next transition (a τ transition due to the scope end event) must lead from $s_{current}(s_14)$ to s_6.

$$\zeta_{seq} = \{27,1\},$$
$$proctrans = \{(s_0,a_0,s_1,2^1,2),(s_1,a_1,s_2,2^1,6),(s_2,a_2,s_3,2^1,7),(s_2,a_3,s_4,2^1,8),$$
$$(s_3,a_4,s_5,2^1+3^9,10),(s_5,\tau,s_6,2^1+3^9,14),(s_6,a_6,s_7,2^1+3^{15},16),$$
$$(s_7,a_7,s_8,2^1+3^{15},17),(s_7,a_8,s_8,2^1+3^{15},18),(s_8,a_9,s_{10},2^1+3^{15}+5^{19},20),$$
$$(s_{10},\tau,s_{11},2^1+3^{15}+5^{19},24),(s_{11},a_{11},s_{12},2^1+3^{15},25),(s_{12},\tau,s_{13},2^1,26),$$
$$(s_4,a_{13},s_{14},2^1+3^27,28),(s_{14},\tau,s_6,2^1+3^27,32)\},$$
$$sval(\zeta_{seq}) = 2^1+3^27,$$
$$s_{current} = s_2,$$
$$M_1^S = \{s_0,s_1,s_2,s_3,s_4,s_5,s_6,s_7,s_8,s_9,s_{10},s_{11},s_{12},s_{13}\},$$
$$M_1^A = \{a_0,a_1,a_2,a_3,a_4,a_6,a_7,a_8,a_9,a_{11},a_{12},a_{13}\}$$

- Events 16–20 are again tracked within the model up until $s_{current} = s_9$.
- Event 21 again introduces a new scope such that $\zeta_{seq} = \{33, 16, 1\}$ and $sval(\zeta_{seq}) = 2^1 + 3^16 + 5^33$.
- Events 22–28 operate like the previous excerpts. We provide the end state after event 28 has been processed.

$\zeta_{seq} = \{\}$,
$proctrans = \{(s_0, a_0, s_1, 2^1, 2), (s_1, a_1, s_2, 2^1, 6), (s_2, a_2, s_3, 2^1, 7), (s_2, a_3, s_4, 2^1, 8),$
$(s_3, a_4, s_5, 2^1 + 3^9, 10), (s_5, \tau, s_6, 2^1 + 3^9, 14), (s_6, a_6, s_7, 2^1 + 3^{15}, 16),$
$(s_7, a_7, s_8, 2^1 + 3^{15}, 17), (s_7, a_8, s_8, 2^1 + 3^{15}, 18), (s_8, a_9, s_{10}, 2^1 + 3^{15} + 5^{19}, 20),$
$(s_{10}, \tau, s_{11}, 2^1 + 3^{15} + 5^{19}, 24), (s_{11}, a_{11}, s_{12}, 2^1 + 3^{15}, 25), (s_{12}, \tau, s_{13}, 2^1, 26),$
$(s_4, a_{13}, s_{14}, 2^1 + 3^{27}, 28), (s_{14}, \tau, s_6, 2^1 + 3^{27}, 32), (s_9, a_{15}, s_{15}, 2^1 + 3^{15} + 5^{33}, 34),$
$(s_{15}, \tau, s_{11}, 2^1 + 3^{15} + 5^{33}, 38)\}$,
$sval(\zeta_{seq}) = 0$,
$s_{current} = s_2$,
$M_1^S = \{s_0, s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8, s_9, s_{10}, s_{11}, s_{12}, s_{13}, s_{15}\}$,
$M_1^A = \{a_0, a_1, a_2, a_3, a_4, a_6, a_7, a_8, a_9, a_{11}, a_{12}, a_{13}, a_{15}\}$

A.2.0.8. Resulting model

As we can see, the behavior of the example has been covered by branch coverage. Because the logs were covering all branches in two passes, we are able to reconstruct the model with two passes of our model reconstruction algorithm as opposed to a reconstruction requiring path coverage, which would have needed four passes. This saving effect is more drastic the more branches exist in the behavior that is reconstructed—path coverage becomes infeasible already with a realistic amount of branches due to exponential combinatorial growth of possible paths that would need to be processed. However, an important disadvantage of this approach is that literally everything that is part of the analysis must be logged. We currently disregard the data-flow and introduce non-determinisms that effectively lead to false positives in the analyses performed later. If we want to reduce these false positives, we have to symbolically log data operations and map them to the target concept. It is not possible to simply evaluate data values at specific behavioral points as the execution history up to each such behavioral point may be different and untraced.

The resulting model from exemplary model reconstruction is depicted in Figure A.1. The actions, transitions, and states that are added in the second pass of the algorithm are marked grey. From this figure, it is clearly visible that branches are always constructed completely also during the first pass due to the look-ahead events. These merely hinted branches will be of use for the test case simulation.

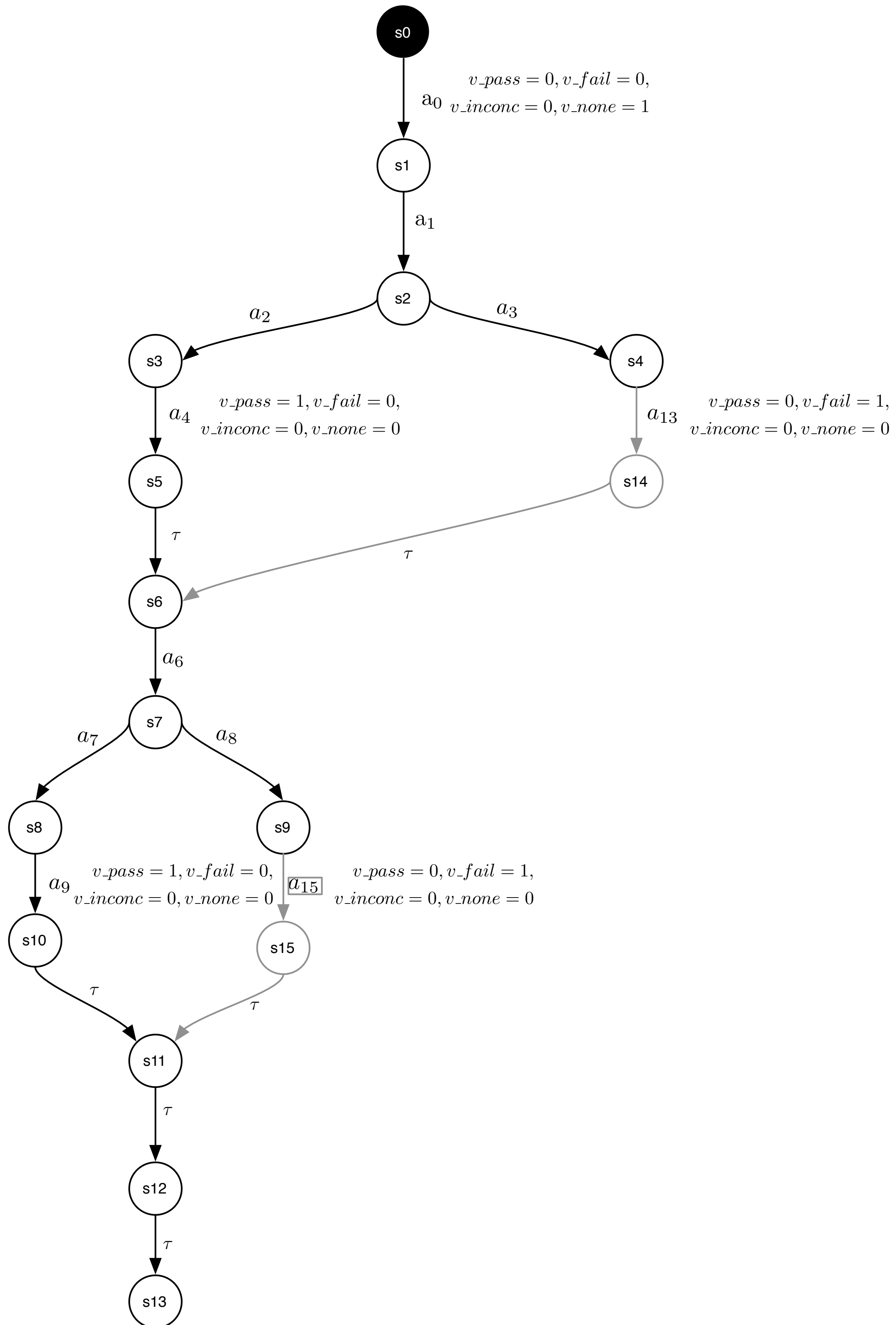


Figure A.1.: Reconstructed Model

A.3. Simulation Coloring

Given colors $C = \{c_1, c_2, c_3\}$ and the relation $COL \subseteq S \times C$, the following three steps provide descriptions for the simulation steps for the model reconstruction algorithm.

Algorithm A.2 (State Processing) Before the model increment reconstruction algorithm alters $s_{current}$ to move to the next state, perform the following steps:

- Add $(s_{current}, c_1)$ to COL.

In the following, we provide the extension that is responsible for choosing new branches in the abstract execution of the test specification.

Algorithm A.3 (Transition Selection) After a sequence of message input events and directly before a new event is processed by the model increment reconstruction algorithm, perform the following steps.

- Find a transition $(s_{current}, a, s') \in \lambda$ where $(s', c_i), 1 \leq i \leq 3 \notin COL$.
- If there is no such transition, then find a transition where $(s', c_2) \notin COL$.
- If there still is no s' selected, then choose an arbitrary s' with $(s_{current}, a, s') \in \lambda$.
- Set $s_{current} \leftarrow s'$.
- Execute a and move forward.
- Add $(s_{current}, c_1)$ to COL.

Finally, we need to reverse track our graph and mark states with the c_3 color.

Algorithm A.4 (Post Processing) The post processing step is essentially where the backtracking is happening. We can follow the executed path backwards by looking which nodes are marked with c_1. When this part of the simulation is executed, $s_{current}$ is located on a final state.

- Loop over the following steps until $s_{current} = s_0$.
  - Add $(s_{current}, c_3)$ to COL.
  - Find a state s' with $(s', a, s_{current})$ where $(s', c_1) \in COL$.
  - If there are transitions $(s', a, s'') \in \lambda$ with $(s'', c_3) \notin COL$, exit the loop.
  - Otherwise, set $s_{current} \leftarrow s'$.
- Recolor all states s with $(s, c_1) \in COL$: find all s with $(s, c_1) \in COL$, for each of these s add (s, c_2) to COL and remove (s, c_1) from COL.

The simulation and model reconstruction is complete when all test components have marked their respective initial states s_0 with c_3.

A.4. The LTSML Metamodel

The LTSML Metamodel is essentially a direct translation of the formal EMIOTS model defined in Ecore [102] and XML Schema [137]. Figure A.2 illustrates the metamodel. In addition to the EMIOTS model, it contains additional information, such as optional color values (for visualization purposes) or line numbers (for mapping back the model checking output to TTCN-3).

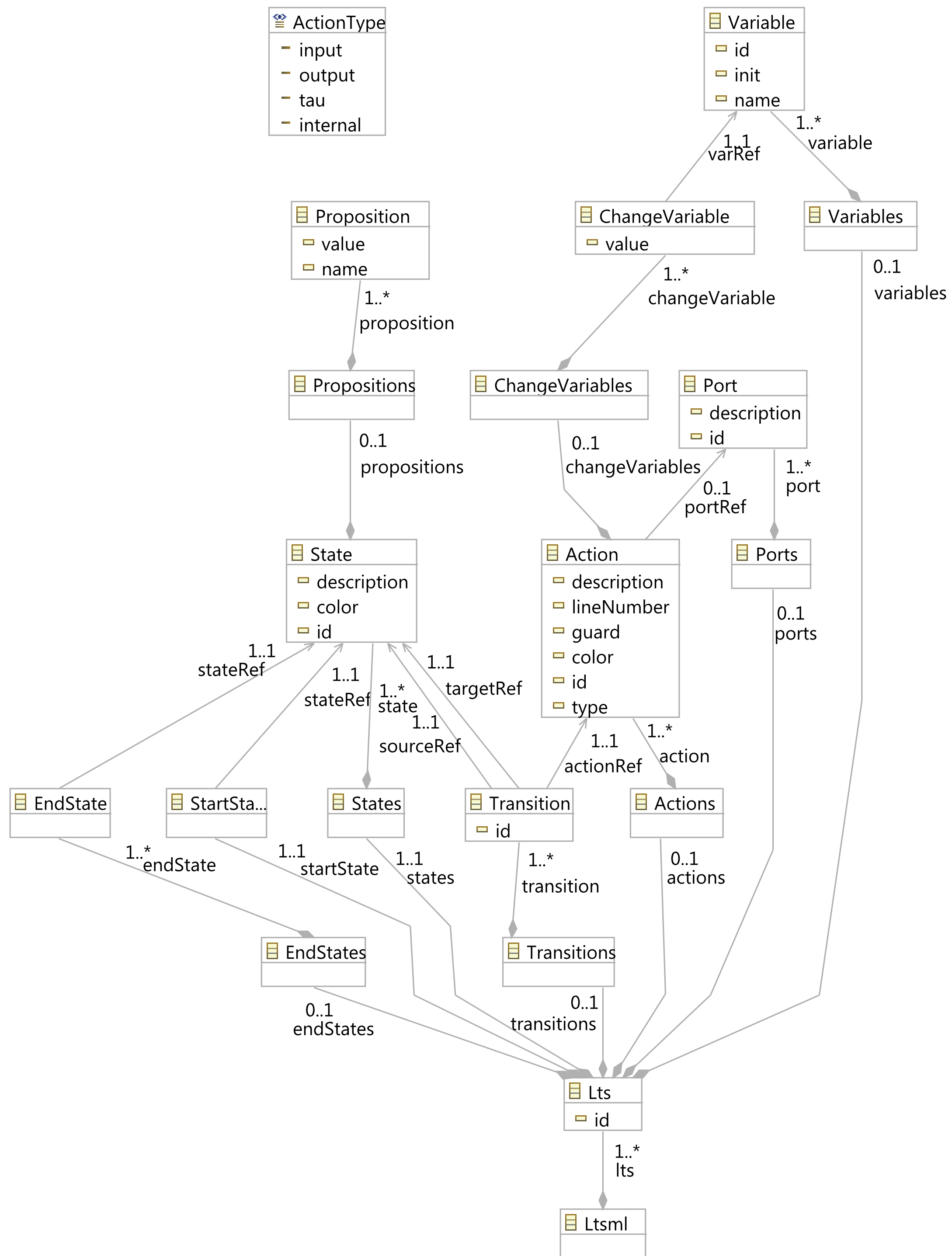


Figure A.2.: LTSML Metamodel

Curriculum Vitae

Benjamin Immanuel Eberhardt Elmar Zeiß

Persönliche Daten

Geburt	27.08.1979 in Frankfurt am Main
Staatsangehörigkeit	Deutsch

Wissenschaftlicher Werdegang

1986–1990	Grundschule Burg auf Fehmarn
1990–1999	Inselgymnasium Burg auf Fehmarn; Abschluss: Abitur
2000–2004	Studium der Angewandten Informatik (Bachelor) an der Georg-August-Universität Göttingen; Abschluss: Bachelor of Science.
2004–2006	Studium der Angewandten Informatik (Master) an der Georg-August-Universität Göttingen; Abschluss: Master of Science.
2006–2010	Doktorand am Institut für Informatik der Georg-August-Universität Göttingen. Gefördert durch ein Doktoranden-Stipendium der Siemens AG München.