

Model-Based Protocol Testing in an ERLANG Environment

Johan Blom



UPPSALA
UNIVERSITET

Dissertation presented at Uppsala University to be publicly examined in ITC 2446, Uppsala, Thursday, 21 April 2016 at 13:15 for the degree of Doctor of Philosophy. The examination will be conducted in English. Faculty examiner: Professor David Whalley (Florida State University).

Abstract

Blom, J. 2016. Model-Based Protocol Testing in an Erlang Environment. *Uppsala Dissertations from the Faculty of Science and Technology* 122. 249 pp. Uppsala: Acta Universitatis Upsaliensis. ISBN 978-91-554-9494-0.

Testing is the dominant technique for quality assurance of software systems. It typically consumes considerable resources in development projects, and is often performed in an ad hoc manner. This thesis is concerned with model-based testing, which is an approach to make testing more systematic and more automated. The general idea in model-based testing is to start from a formal model, which captures the intended behavior of the software system to be tested. On the basis of this model, test cases can be generated in a systematic way. Since the model is formal, the generation of test suites can be automated and with adequate tool support one can automatically quantify to which degree they exercise the tested software.

Despite the significant improvements on model-based testing in the last 20 years, acceptance by industry has so far been limited. A number of commercially available tools exist, but still most testing in industry relies on manually constructed test cases.

This thesis address this problem by presenting a methodology and associated tool support, which is intended to be used for model-based testing of communication protocol implementations in industry. A major goal was to make the developed tool suitable for industrial usage, implying that we had to consider several problems that typically are not addressed by the literature on model-based testing. The thesis presents several technical contributions to the area of model-based testing, including

- a new specification language based on the functional programming language Erlang,
- a novel technique for specifying coverage criteria for test suite generation, and
- a technique for automatically generating test suites.

Based on these developments, we have implemented a complete tool chain that generates and executes complete test suites, given a model in our specification language. The thesis also presents a substantial industrial case study, where our technical contributions and the implemented tool chain are evaluated. Findings from the case study include that test suites generated using (model) coverage criteria have at least as good fault-detection capability as equally large random test suites, and that model-based testing could discover faults in previously well-tested software where previous testing had employed a relaxed validation of requirements.

Keywords: Testing, Erlang, Model-Based Testing, Symbolic Execution, Communication Protocols

Johan Blom, Department of Information Technology, Division of Computer Systems, Box 337, Uppsala University, SE-75105 Uppsala, Sweden.

© Johan Blom 2016

ISSN 1104-2516

ISBN 978-91-554-9494-0

urn:nbn:se:uu:diva-279489 (<http://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-279489>)

To Anna and Emilia

Contents

1	Introduction	21
1.1	Testing	22
1.2	Terminology	24
1.3	Introducing state machines	25
1.4	Erlang	28
1.5	Model-based testing	29
1.5.1	Creation of a formal model	30
1.5.2	Validation of the formal model	31
1.5.3	Test Suite Generation	32
1.5.4	Concretization	34
1.5.5	Execution of test cases	35
1.5.6	Verification	35
1.6	Contributions of this Thesis	35
1.6.1	Modeling Language	37
1.6.2	Specifying Test Case Selection	37
1.6.3	Efficient Generation of Test Suites	38
1.6.4	Concretization of Generated Test Cases	39
1.6.5	Efficient Test Execution and Verification	39
1.6.6	Evaluation of Different Test Generation Strategies	40
1.7	Organization of Thesis	41
2	A specification language based on <code>ERLANG</code>	42
2.1	Introducing <code>ERLANG/EFsm</code>	42
2.2	An Overview of <code>ERLANG/EFsm</code>	44
2.3	<code>ERLANG/EFsm</code> - an extension of <code>ERLANG</code>	49
2.3.1	Syntax for a restricted set of <code>ERLANG</code>	49
2.3.2	<code>ERLANG/EFsm</code> extensions to <code>ERLANG</code>	55
2.4	Operational semantics of <code>ERLANG/EFsm</code>	62
2.4.1	Pattern matching	63
2.4.2	Transition Rules for <code>ERLANG/EFsm</code> Expressions	63
2.5	Derived State Machines	68
2.6	Runs, Traces, and Test Cases	70
2.7	Symbolic operational semantics	70
2.7.1	Pattern matching	73
2.7.2	Transition Rules for <code>ERLANG/EFsm</code> Expressions	73
2.8	Correspondence Between non-Symbolic and Symbolic Semantics	79

2.9	Symbolic Runs and Test Cases	81
2.10	Defining and Normalizing Edge Clauses	83
2.11	Creating an executable specification	84
2.11.1	The <code>gen_fsm</code> behavior in <code>ERLANG/OTP</code>	85
2.11.2	Creating an executable <code>ERLANG</code> module	85
3	Specifying test case selection	91
3.1	Observers: An Informal Introduction	92
3.2	<code>ERLANG/OBS</code> - observers with <code>ERLANG</code> syntax	95
3.2.1	Syntax for <code>ERLANG/OBS</code>	98
3.3	Defining observer predicates	101
3.3.1	Match variables	102
3.3.2	Match functions	102
3.3.3	Observer predicate definition in <code>ERLANG/OBS</code>	102
3.4	Graphical observer notation	107
3.5	Operational semantics of <code>ERLANG/OBS</code>	108
3.5.1	Observer expressions	109
3.5.2	Observer predicates	110
3.5.3	Observer edge clauses	119
3.6	The Observer Defined by an <code>ERLANG/OBS</code> Specification	120
3.7	Symbolic semantics of <code>ERLANG/OBS</code>	121
3.7.1	Observer expressions	123
3.7.2	Observer predicates	123
3.7.3	Observer edge clauses	127
3.7.4	Correspondence with Operational Semantics	128
3.8	Symbolic Observers and Symbolic Coverage	129
4	Coverage criteria	134
4.1	Model-independent coverage criteria	134
4.1.1	Coverage of guards	134
4.1.2	Coverage of locations	138
4.1.3	Coverage of paths	138
4.1.4	Coverage of data flow	139
4.2	Model-dependent coverage criteria	142
5	Generating test suites	146
5.1	Generating Symbolic Test Cases	147
5.2	Generating symbolic test suites	150
5.2.1	Refining the search exploration algorithm	152
5.3	Efficiently representing sets of states	152
5.3.1	Bitvector Representation of Sets of Observer States	155
5.4	Concretisation	156
5.4.1	Generating abstract test suites from symbolic test suites	157

5.4.2	Generating concrete test suites from abstract test suites	158
5.5	Alternative usage of observer automata	159
5.5.1	Filtering the specification	159
5.5.2	Model Checking of a Specification	160
6	Introducing ERLY MARSH	162
6.1	Prototyper and Simulator	162
6.2	Model compiler	164
6.3	Pretty printer	165
6.4	Test suite generator	166
6.5	Test suite execution tool	167
6.6	ERLY MARSH Verifier	169
6.7	Test suite report tool	169
7	Evaluation: Testing a Telecom Software Application	172
7.1	Mobile Arts Advanced Mobile Location Center	173
7.1.1	The A-MLC ERLANG /EFSM specification	175
7.1.2	Symbolic test suite generation	179
7.1.3	Test Execution Environment	180
7.2	Independent variables	183
7.2.1	Symbolic test case selection techniques	185
7.2.2	Abstract test case selection techniques	185
7.2.3	Test execution strategies	188
7.2.4	Base models	189
7.3	Dependent variables	189
7.3.1	Failures	190
7.3.2	Faults	191
7.3.3	Source code coverage	193
7.3.4	Abstract test suite size	194
7.3.5	Execution time	194
7.4	Threats to validity	195
7.4.1	Internal validity	195
7.4.2	External validity	195
8	Results using ERLY MARSH on A-MLC	197
8.1	Summary of the results	197
8.1.1	Coverage based test case selection	198
8.1.2	Random test case selection	198
8.1.3	Manual test case selection	200
8.1.4	Testing with projected specification and reduced validation	201
8.2	Failures found while testing	202
8.3	Faults found while testing	204
8.3.1	Characteristics of a selection of test suites	208

8.4	Code coverage	210
8.5	Test suite size and execution times	214
8.6	Summary of main findings	216
8.6.1	Experiences from the Case Study	217
9	Other tools for testing <code>ERLANG</code> programs	219
9.1	<code>QUICKCHECK</code> and other random based testing tools	219
9.2	<code>DIALYZER</code> static analysis tool	220
10	Related work	223
10.1	Test suite generation techniques	223
10.2	Coverage criteria and Test purposes	225
10.3	Case studies	227
11	Conclusions	230
11.1	Summary	230
11.2	Modeling Language	231
11.3	Specifying Test Case Selection	232
11.4	Efficient Generation of Test Suites	233
11.5	Concretization of Generated Test Cases	234
11.6	Efficient Test Execution and Verification	234
11.7	Evaluation of Different Test Generation Strategies	235
11.8	Discussion	235
11.9	Future Work	236
11.9.1	Creating specifications	237
11.9.2	Generating test suites	237
	References	239

List of Tables

Table 1.1: Test suite that aligns with a 2-wise combination strategy. for a system with 3 parameters with domains $p_1, p_2 \in \{1, 2, 3\}$ and $p_3 \in \{1, 2\}$	34
Table 2.1: <code>ERLANG</code> syntax base.	50
Table 2.2: A restricted set of the <code>ERLANG</code> syntax.	51
Table 2.3: <code>ERLANG/EFSM</code> syntax extensions to <code>ERLANG</code>	56
Table 3.1: <code>ERLANG/OBS</code> syntax.	99
Table 3.2: <code>ERLANG/OBS</code> observer predicate definition syntax.	103
Table 3.3: Observer predicates defined by pattern matching with a match variable.	106
Table 8.1: Summary of test results for coverage based test case selection, using the original and normalized base models for A-MLC.	199
Table 8.2: Summary of test results for creating and executing a selection of the random test suites on the A-MLC.	200
Table 8.3: Summary of test results on the A-MLC for manual test suite.	201
Table 8.4: Summary of test results found with test suites based on a reduced base model, and using limited validation.	202
Table 8.5: Failures found when running test suites generated from the original base model.	203
Table 8.6: Classification of executable statements reported <i>not</i> covered by the tool <code>COVER</code> . The percentages shown are the part of the complete module not covered because of the given reason.	212

Acknowledgments

Once upon a time I started my PhD studies with the ambition to finish really fast! That did not happen. In fact I have spent quite a number of years with this PhD thesis. Still I have never regretted I did start, as most of the time I have simply enjoyed myself with empowering discussions in the friendly atmosphere at the IT department at Uppsala University.

My supervisor during all these years have been Bengt Jonsson. Naturally, without his endless passion, ideas and patience I would never have finished.

In such a number of years the number of people that, in some way or another, has inspired me, helped me, or just been there for me are numerous. Thus, even if not listed below, you are not forgotten, a big thank you for everything!

But to start from the beginning, my family: mother Evy, father Henry and brother Anders with family that always has supported me and to whom I'm utterly grateful. Dad, you had always valued people that never gave up high. I'm sure you would be proud now, just wish you would still be with us!

Music brought me to Uppsala. In fact, I would probably not even have started studying at the University if not for that student music festival in Linköping 1986 that completely blew my mind at the time.

From the times when I started my PhD studies I have great memories of the co-author of my first articles Lars Kempe and my, at the time, secondary supervisor Roland Bol. Later, when starting all over again, Anders Hessel and Paul Petterson played important roles. Paul, beginning his PhD studies approximately in the same time as I did, now acting as secondary supervisor. I must also mention Lars-åke Fredlund that always have been very friendly and supportive whenever I have asked for help. Among the many colleagues at Mobile Arts that during the years have encouraged my continued work with the thesis Lars Kari deserves special mentioning.

There were periods that family life and starting up a company simply became to overwhelming to sometimes even think about thesis work. When slowly recovering from those periods it really meant a lot for me that Bengt Ahlgren was so kind to let me use some space at SICS for working with the thesis.

Sven-Olof Nyström, my current secondary supervisor, proved an invaluable help, especially during the last frantic part of this thesis creation.

With this thesis finished, this period of my life comes to an end and a new era begins. So finally, my love, Pernilla, mother of my two children and always my greatest supporter. Love you deep and dearly! Let us now, together, make all those dreams come to life.

This work was supported in part by the Swedish Innovation Agency (VINNOVA) within the ASTEC competence center, and in part by the Swedish Research Council (VR) within the UPMARC Linnaeus centre of Excellence.

Publications by the Author

This thesis is based on the following papers:

- I Johan Blom and Bengt Jonsson
Automated Test Generation for Industrial ERLANG Applications.
In Proc. 2003 ACM SIGPLAN workshop on Erlang, pages 8–14,
Uppsala, Sweden, Aug. 2003.
- II Johan Blom, Anders Hessel, Bengt Jonsson, and Paul Pettersson
Specifying and Generating Test Cases Using Observer Automata.
In Proc. FATES, 4th. International Workshop on Formal Approaches
to Testing of Software, volume 3395 of *Lecture Notes in Computer
Science*, pages 125–139. Springer-Verlag, 2004.
- III Johan Blom, Bengt Jonsson, and Sven-Olof Nyström
*Industrial Evaluation of Test Suite Generation Strategies for Model-
Based Testing.*
In *A-MOST '16: Proceedings of the 12th international workshop on
Advances in Model-Based Testing*. IEEE, 2016.

The author has also done some earlier work on modeling and formalization of requirements:

- I Johan Blom, Bengt Jonsson, and Lars Kempe
Using Temporal Logic for Modular Specification of Telephone Services.
In *Feature Interactions in Telecommunications Systems*. IOS Press,
Amsterdam, Netherlands, May 1994.
- II Johan Blom, Roland Bol, and Lars Kempe
Automatic Detection of Feature Interactions in Temporal Logic.
In *Feature Interactions in Telecommunications Systems III*. IOS
Press, Amsterdam, Netherlands, October 1995.
- III Johan Blom, Roland Bol, Bengt Jonsson, and Jan Nyström
Creation of Dependent Features.
In *Proceedings of RVK '96, Radio Vetenskap och Kommunikation'96*,
Luleå, Sweden, June 1996.
- IV Johan Blom and Bengt Jonsson
Constraint Oriented Temporal Logic Specification.
In *Formal Systems Specification, The RPC-Memory Specification
Case Study*, volume 1169 of *Lecture Notes in Computer Science*,
pages 161–182. Springer-Verlag, 1996.

V Johan Blom

Formalisation of Requirements with Emphasis on Feature Interaction Detection.

In *Feature Interactions in Telecommunications Systems IV*, pages 61–77. IOS Press, Amsterdam, Netherlands, 1997.

Summary in Swedish

Att “datorn” ibland gör fel och inte fungerar är något vi alla lärt oss och tvingats anpassa oss efter. Inte utan problem dock. Många gånger är konsekvenserna också väldigt kostsamma och utgör i värsta fall personfara. Den mest använda tekniken för att kvalitetssäkra system är testning. Med testning menas här verifiering för att hitta fel, buggar och andra problem genom att på ett givet indata kontrollera mot en specifikation av systemet att utdata är det förväntade. Ett system kan här vara en mjukvarubaserade komponent i ett komplett system eller ett komplett system i sig. Givet den stora mängd med komplicerade system som utvecklats, sedan många år tillbaks, kan det tyckas förvånande att testning i praktiken ofta utförs ineffektivt och till stora kostnader. Förklaringen kan antas vara att med mer komplexa system så blir också testandet mer komplext.

Den här avhandlingen handlar om *modellbaserad testning*, en teknik för att automatisera och förbättra effektiviteten i testningen. I modellbaserad testning utgår man från en abstrakt formell specifikation av ett system som man är intresserad av. Idén är att denna specifikation inte är en komplett beskrivning av systemet utan fångar just de egenskaper man är intresserad av att testa. En specifikation är en modell och eftersom modellen är formell kan testfall genereras automatiskt. Från en stor och komplicerad modell kan antalet möjliga testfall bli så stort att det inte är möjligt att testa alla. Detta är speciellt vanligt för protokoll, då modellerna kan representeras med tillståndsmaskiner. Man måste då välja en delmängd av alla möjliga testfall. Ett vanligt sätt är att ange ett *täckningskriterium* som måste uppfyllas av en testsvit. Exempel inkluderar tex alla tillstånd eller alla kanter i modellen. Genom adekvat verktygsstöd är det då möjligt att generera mycket stora testsviter där man också kan kvantifiera över hur den genererade testsviten förhåller sig till systemet man vill testa. Forskning inom detta område har med framgång använts i industriella system och gett upphov till algoritmer för testsvitsgenerering och verktyg som tex TorX [Tretmans 03], Gotcha [Friedman 02], Conformiq Designer [Huima 07], och Spec Explorer [Veanes 08]. Men fortfarande så baserar sig den mesta testningen i industrin på manuellt skapade testsviter.

Arbetet med denna avhandling motiverades från början på högt ställda krav på korrekt beteende i mobila telekom nätverk och på bristande tillgång till resurser för testning i det lilla företaget. Produkten som utvecklades används för att bestämma tex positionen hos en mobiltelefon i ett mobilnät. En typisk användare är en SOS-central som med detta system kan lokalisera nödställda. För testning av denna produkt utvecklades ett verktyg för modellbaserad testning av kommunikationsprotokoll.

Den här avhandlingen innehåller flertal tekniska bidrag inom modellbaserad testning av vilka de huvudsakliga är

- ett nytt specifikationspråk baserat på de funktionella programmeringsspråket `ERLANG`,
- ny teknik för specificering av täckningskriterium vid testfallsgenerering, och
- en teknik för att automatiskt generera testsviter.

Baserat på detta implementerades ett verktyg med vilket man kan både generera och exekvera testsviter från specifikationspråket. Avhandlingen innehåller också en större fallstudie där de tekniska bidragen och verktyget utvärderas. Nedan följer en närmare genomgång av dessa bidrag.

Ett välkänt problem med modellbaserad testning är att kostnaden för att skapa modellen kan antas vara hög. I denna avhandling fokuseras främst på specifikationer av protokoll i form av tillståndsmaskiner. En viktig komponent i detta sammanhang är valet av specifikationspråk i vilket det ska vara möjligt att uttrycka önskade egenskaper i systemet kort och koncist. För att möjliggöra effektiv analys är det också viktigt att språket är formellt. Baserat på det funktionella programmeringsspråket `ERLANG` och med egenskaper såsom dynamisk typning och mönstermatchning introduceras för detta ändamål `ERLANG/EFM`. För detta språk ges, förutom syntax, också en operational semantik. Vår förhoppning är att detta (lilla) språk ska vara lätt att ta till sig för både testare och utvecklare. Speciellt, naturligtvis, för redan förtrogna med `ERLANG`.

Eftersom olika testsviter är lämpliga för olika situationer är det önskvärt att ett testgenereringsverktyg har en god flexibilitet i att generera olika täckningskriterium. För att som användare fritt kunna specificera täckningskriterium introducerar vi *observatörer* och en notation för att formellt kunna uttrycka dessa. Observatörer kan ses som en separat komponent som observerar exekveringen av en tillståndsmaskin samtidigt som den kommer ihåg vilka syntaktiska element som täckts av exekveringen. För specifikation av observerare introduceras `ERLANG/OBS`, även detta med `ERLANG` syntax.

Med en stor och komplicerad modell kan också representationen av testfallen bli ett problem. Vi hanterar detta genom *symbolisk* exekvering av modellen vilket medför att vi effektivt kan representera mängder av testfall med enstaka exekveringar. Observatörer måste i detta sammanhang hantera symboliska parametrar och man kan använda sig av olika strategier för att instansiera dessa parametrar.

För att utvärdera idéerna om modellbaserad testning skapades ett verktyg, `ERLY MARSH`, som använts i en större fallstudie. Systemet som testats, A-MLC, används för att bestämma tex positionen av en mobiltelefon i ett mobilnät. För att generera testfall till detta system skapades en specifikation skriven i `ERLANG/EFM`. Från specifikationen genererades sedan symboliska testsviter med hjälp av en mängd täckningskriterier uttryckta med observatörer (och skrivna i `ERLANG/OBS`). Från dessa symboliska testfall valdes abstrakta testfall, från vilka det i sin tur valdes konkreta

testfall som gick att exekvera mot en instans av A-MLC. Vi lyckades då hitta ett antal fel i implementationen av A-MLC som vi sedan använder för att jämföra effektiviteten hos de olika genererade testsviterna, mätt i antalet hittade buggar och kodtäckning. Vidare jämför vi resultaten från testsviterna genererade från olika täckningskriterier med en mindre mängd manuellt skapade testfall och en större mängd slumpvis utvalda testfall. Resultatet från denna fallstudie indikerar att

- ett ganska enkelt täckningskriterium som “alla kanter i tillståndsmaskinen” ger förhållandevis bra resultat med hänsyn tagen till antalet testfall inkluderade i testsviten,
- det potentiellt inte finns några stora skillnader mellan resultaten från testsviter genererade från täckningskriterium och slumptestning om man tar hänsyn till storleken, och
- med ökad storlek på testsviterna så minskar skillnaderna mellan antalet hittade buggar och antalet rader kod som täcks.

1. Introduction

Testing is a dominant technique for quality assurance of software systems. Testing also consumes significant amounts of resources in development projects [Myers 79, Beizer 90, Sommerville 10]. Given this, one would expect that software testing would have developed into an exact science with carefully worked out routines for guaranteeing the absence of errors in software. Unfortunately, this is not the case. Testing still remains more of an art than a science. One of the reasons is that software systems are becoming increasingly complex, and so is testing them. Hence, it is of general interest to refine existing test methods and find possible ways of improving the testing activities.

This thesis is concerned with Model-Based Testing, which is an approach to make testing more systematic. The general idea in model-based testing is to start from a formal model, which captures the intended behavior of the software system to be tested. A software system should here be interpreted in a broad sense and can be a component in a larger system or the complete system. Ideally, this model should be created early in the development process, possibly during or even before the system is implemented. On the basis of this model, test cases can be generated in a systematic way. Since the model is formal, the generation of test suites can be automated, and it is possible to generate test suites (i.e., sets of test cases) that exercise all functionality of the tested software. With adequate tool support one can automatically generate very large test suites, and automatically quantify to which degree they exercise all of the tested software.

An important application area in model-based testing is that of testing communication protocols and other similar classes of reactive systems. For such systems, detailed behavioral specifications can be used as models, and serve both as a basis for test suite generation, and as an oracle for assigning verdicts to test cases. Research in this area has given rise to automatic test suite generation algorithms and tools like TorX [Tretmans 03], Gotcha [Friedman 02], Conformiq Designer [Huima 07], and Spec Explorer [Veanes 08] among others, which have also been used successfully on industrial-size systems.

Despite the significant developments in model-based testing, [El-Far 02, BJK 04, Utting 07, Marinescu 15] in the last 20 years, its impact on industrial practice has still been rather limited. A number of commercially available tools exist, but still most testing in industry relies on manually

constructed test cases. The work on this thesis was originally motivated by the high requirements on software quality in telecom networks and the scarce resources for testing available in a small software development company. An application was developed and there was an identified need to systematically test this software application in a more rigorous manner. The application developed is often referred to as a location center and used by e.g., emergency centers, to provide details about the location of mobile devices in case of emergency. We therefore started to develop automated tool support for model-based testing of communication protocols. A major goal was to make the developed tool suitable for industrial usage, implying that we had to consider several problems that typically are not addressed by the literature on model-based testing.

This thesis describes a methodology and associated tool support, which is intended to be used for model-based testing of communication protocol implementations in industry. It presents several technical contributions to the area of model-based testing, of which the main ones are

- a new specification language based on the functional programming language `ERLANG`,
- a novel technique for specifying coverage criteria for test suite generation, and
- a technique for automatically generating test suites.

Based on these developments, we have implemented a complete tool chain that generates and executes complete test suites, given a model in our specification language. The thesis also presents a substantial industrial case study, where our technical contributions and the implemented tool chain are evaluated.

In the remainder of this introduction, we will in Section 1.1 outline a few different purposes to test, in Section 1.2 introduce basic concepts in testing, in Section 1.3 introduce state machines, in Section 1.4 introduce `ERLANG`, and in Section 1.5 introduce model-based testing. Finally, in Section 1.6 we discuss the contributions, and in Section 1.7 outlines, of the rest of this thesis.

1.1 Testing

Testing is an investigation conducted to provide stakeholders with information about the quality of the product or service under test. Software testing can also provide an objective, independent view of the software to allow the stakeholders understand e.g., risks involved of a software implementation. After testing we have gained increased confidence in the quality and correctness of the system.

Testing can be performed during many different phases of the software lifecycle, and must be adapted accordingly. Let us give a brief account of

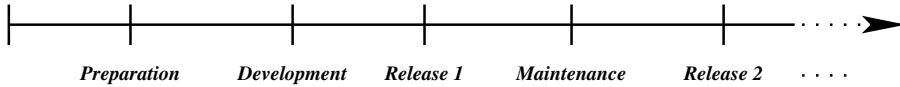


Figure 1.1. Simplified description of the life-cycle of a larger software system.

the different forms of testing, based on a simplified description of the software lifecycle, graphically depicted in Figure 1.1, in which actual software development is preceded by a preparation phase (where requirements are collected and evaluated) and succeeded by a series of software releases and management phases.

During the preparation phase, there may be requirements on external components, such as database, operating system, etc., which must be tested before building the system has even started. We may then need to *compatibility test* and *performance test* involved components. During the actual development phase, the developers subject individual software components to *unit tests* and *component tests*, and thereafter *integration test* the combination of these components. After development is finished, when it is time to release a system to a customer, the bulk of the testing effort typically occurs. Different forms of testing include:

- *System testing*, i.e., testing of the behavior of the whole system in an environment which corresponds to the target environment. System testing checks both functional requirements, to ensure logical and temporal correctness, and non-functional requirements, to ensure e.g., high availability and correct behavior under heavily utilization.
- *Install/uninstall testing*, i.e., testing whether it is possible to build, install, upgrade etc. the system on the target machines.
- *Acceptance testing*, i.e., testing to verify if the system meets the customer's specified requirements. The goal is to establish confidence in the system. Finding faults is not the the main focus.

Whenever a system has been released, delivered and accepted by a customer, it may still happen that faults are revealed that previous testing has not discovered. Then, maintenance of the system is necessary including further testing so that the faults can be removed. To make another, new, release of the system we may then only need to *regression test* the system after changes have been made. This means to re-test the system, using a selected subset of test cases. Naturally, in general, maintenance may also involve development of new features that has to be tested as such.

1.2 Terminology

The *System Under Test (SUT)* is the complete system that one wants to test. Note that testing need not exercise all functionality of the SUT: for example, in the evaluation in Section 7 of this thesis, testing concerns only the part of the SUT that is reachable via a set of protocol interfaces.

A fundamental distinction is that between white-box and black-box testing. *White-box testing* is based on the structure of the source code of the SUT. *Black-box testing* is based on the requirements on the functionality of the SUT and assumes no knowledge of the implementation. For example, a test suite for white-box testing may include test cases that depend on the usage of variables in the source code of the SUT. A test suite for black-box testing may include test cases that depends on requests sent to the SUT and responses from the SUT. In this thesis, we focus on black-box testing.

Since it is not feasible to test all possible executions of the SUT, some selection of “what to test” is necessary. A *test purpose* is a specific goal or property deemed necessary to test. In a communication protocol, a test purpose might be to check that a communication can be established. Ideally, all interesting functional requirements of the specification should be captured in the form of test purposes.

Test purposes are used to derive *test cases*. Each test case describes a designated starting state, a sequence of test inputs to be applied, and an expected response from the SUT. A *test suite* is a set of test cases. *Test suite generation* is the process of selecting a set of test cases to form a test suite. *Test suite execution* is the process of executing test cases part of a test suite. A *test oracle* [Richardson 92] predicts the expected response from the SUT when executing test cases. The test oracle reports a *test verdict* for each executed test case. Ideally, the test verdict should be “pass” or “fail” but can also be “inconclusive” if the test oracle can not determine conclusively whether the output of the SUT is correct. A *failure* is an observed deviation in the behavior of the SUT, i.e., the test oracle reports “fail”. A *fault* is the cause of such a failure.

The term *coverage* denotes some measure of thoroughness of a test suite. If the source code is available (as in white-box testing) we can measure the thoroughness of a test suite by e.g., calculating how many percent of the lines of source code of the SUT that are executed by the test suite. Other possible measures include the number of variable definitions that are actually used, or the number of paths in the control flow of the source code (e.g., of a certain length) that are exercised. A *coverage criterion* denotes a property that should be satisfied to some degree by the test suite. Since different coverage criteria are suitable in different situations, and differ in fault detection capability, cost, information about where

potential faults may be located, etc., it is highly desirable to be able to generate test suites for a wide variety of different coverage criteria.

In order to actually execute a test suite against a SUT, a *test harness* (or automated test execution framework) is needed. This is a collection of software and test data configured to test a SUT by running it under varying conditions and monitoring its behavior and outputs. A test harness should handle all the following phases when executing a test suite.

- Setup of test suite, i.e., achieve any necessary precondition necessary before a test suite can be executed. This may e.g., include starting a run-time environment and the SUT, and ensuring that they are ready to start executing test cases.
- Setup of test case, i.e., achieve any necessary precondition necessary before a particular test case can be executed. This may include preparing a run-time environment with expected responses and configuration of the SUT for a particular test case.
- Cleanup, i.e., perform any necessary postcondition necessary to reset any performed preconditions, This may include resetting a run-time environment and SUT to an initial state, e.g., by removing configuration associated to an executed test case.
- Collection and presentation of test verdicts, making it easy for humans or computer programs to understand and to process the test results.

1.3 Introducing state machines

Protocols typically have a number of states that can be captured in a state machine. As background, we will therefore give an informal introduction to state machines. Later, in Section 2, we will introduce a new specification language, based on the **ERLANG** programming language for the specification of state machines.

We assume that a System Under Test (SUT) interacts with its environment through *events*. Events are either *input events*, typically representing the receipt of messages, or *output events*, typically representing the transmission of messages. At any point in time, the state machine is in some *state*. Whenever the SUT receives an input event, it reacts by performing some local computation, thus the state machine changes its state, and emitting a (possibly empty) sequence of output events. Initially, the state machine is in some designated *initial state*.

To specify a state machine, one must thus specify its input and output events, its set of states, and how it reacts to input events in each given state. Typically, input and output events can be represented as terms of form $a(d_1, \dots, d_k)$, where a is an (input or output) *event type*, and d_1, \dots, d_k are data values from prespecified domains. The set of states

is typically specified by defining a finite set of *locations* and a finite set of *state variables* (with specified domains). Each state is then given by a location and a mapping from state variables to values. The reaction to input events is described by a set of *transitions*. Each transition has designated source and target locations, and consists of a triggering input event type, an optional guard, and a description of the associated internal computation. We illustrate this by an example.

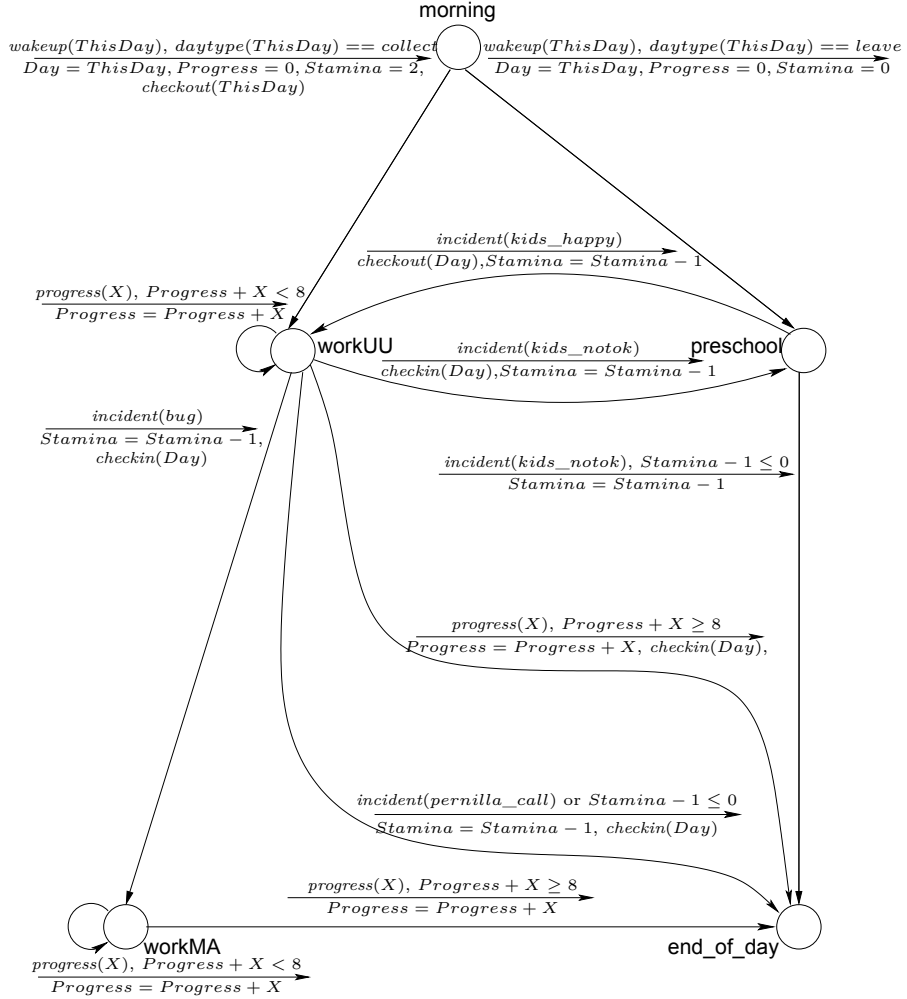


Figure 1.2. A graphical representation of the state machine described in Example 1.1. The complete ERLANG/EFM specification is given in Figure 2.2.

Example 1.1 A state machine which models a typical day in life is represented in Figure 1.2. The state machine accepts three forms of input events:

- *wakeup*(*TDay*) representing a wakeup call where *TDay* is a day in the week,
- *progress*(*X*) representing progress made on work during some period in a day where *X* is an integer in the range 1, ..., 3, and
- *incident*(*I*) representing some externally triggered interception of work, where *I* is the type of an incident. Possible values on *I* are *bug*, *kids_notok*, *kids_happy* and *pernilla_call*.

The output events are of form:

- *checkout*(*Day*) representing a checkout of the latest revision from a server, and
- *checkin*(*Day*) representing a check in of latest work to a server

where, for both output events, *Day* is a day in the week. There are five locations, corresponding to the circles in Figure 1.2. The initial location, **morning**, represents the beginning of a new day, **workUU** represents thesis work at Uppsala University, **workMA** represents work at Mobile Arts, **preschool** represents visiting kids at school, and **end_of_day** represents the end of the day. Three state variables are used: *Day* whose value is the current day in the week, *Stamina*, the stamina left for thesis work, and *Progress*, the total progress made so far during this day. We assume a function *daytype* which, given a day, returns the type of that day, being either *collect* or *leave*. The transitions are represented by edges in Figure 1.2. Each edge is annotated by a label, which above the arrow contains the triggering event type (with formal parameters), and an optional guard (following after a comma). Below the arrow is a statement (in pseudocode), which describes the triggered local computation. For instance, the top left edge describes that whenever the current location is **morning**, an event of form *wakeup*(*TDay*) occurs, and the *daytype* of *TDay* is *collect*, then the state variable *Day* is bound to the value of the parameter of the input event (i.e., the actual value of *TDay*), the state variable *Progress* is bound to 0, the state variable *Stamina* is bound to 2, the output event *checkout*(*TDay*) will be emitted, and the machine will move to location **workUU**.

Let us give an informal overview of the behavior of the state machine. Initially, only a *wakeup*(*TDay*) event may occur. Depending on the type of day the children must either be taken to the preschool (moving to the **preschool** location), or work can start directly (moving to the **workUU** location). Before entering the **workUU** location a *checkout*(*Day*) event is emitted to access the latest work of coworkers. Correspondingly, after leaving the **workUU** location we must emit a *checkin*(*Day*) event to share the latest work. If an unfortunate incident occurs at the preschool (represented by an *incident*(*kids_notok*) input event) this must be im-

mediately taken care of, so we rush to the preschool (**preschool** location) and stay there until the childrens are ok. If we are lucky, the *incident(kids_happy)* event occurs and we may proceed back to thesis work again (the **workUU** location) if we have stamina enough. However, if the childrens situation does not improve or we are out of stamina, we must call it end of day and go home. While in the **workUU** location, or **workMA** location, we may do some progress in work, represented by a *progeess(X)* input event expression, for some X in the range $1, \dots, 3$.

If an incident occurs at Mobile Arts (*incident(bug)* input event), while at the **workUU** location, this must immediately be taken care of at Mobile Arts (the **workMA** location) until we have made enough progress ($Progress \geq 8$) and the day ends (**end_of_day** location).

We will assume that state machines representing models are *deterministic*, meaning that for each state and input event exactly one reaction is possible. If, for some state and some input event, the state machine does not specify a reaction, this means that the corresponding behavior is unspecified. The modeled SUT may in this case exhibit a run-time error or some other behavior that is not described by the model. We will not be interested in generating test cases for such unspecified behavior.

1.4 Erlang

ERLANG is a general-purpose language that supports fault-tolerant, distributed, and concurrent programming. The sequential subset of **ERLANG** is a functional language, with strict evaluation order, single assignment, and dynamic typing. Originally it was a proprietary language within Ericsson, but released as open source 1998.

In this thesis we will use a subset of **ERLANG** (essentially the sequential subset) for specification. Given such “**ERLANG**” specifications we will generate test suites. We will also present a case study in Section 7 on an **ERLANG** based implementation. The techniques presented is not in any sense limited to implementations in **ERLANG** and can be equally well applied on systems implemented in any other language. However, it has been reported by e.g., [Cronqvist 04], studying 150 trouble reports from function testing and system testing, that most faults found in **ERLANG** implementations are misunderstandings of the requirements on a functional level. This makes **ERLANG** based implementations particularly interesting for further studies with the techniques presented in this thesis.

As **ERLANG** is a functional language it enables the user to write programs in a more succinct and abstract way than in imperative languages. It was designed to be suitable for programming communication software at a high level of abstraction, with features such as pattern matching, dynamic

typing, and single assignment variables that allow to write compact programs. Particularly in development of complex control software systems, for example telecom networks, it is important that the implementation language captures system level descriptions in a concise manner.

1.5 Model-based testing

Model-based testing aims to make testing more systematic and more automated. Model-based testing starts from a model, which serves as the basis for systematically generating test cases. Often, the model is a formal description of the SUT's desired behavior. This formal model plays several roles; as a test oracle or as a source for generating test suites. With a formal model we also have means to generate “good” test suites as we can use the model to control the selection of test cases. In this way, model-based testing has the potential to automate a large part of test suite development.

One may argue that all forms of testing are necessarily model-based, since construction of test cases presupposes some mental model of what the SUT is supposed to do [Binder 99], but the idea of model-based testing is to replace implicit mental models by explicit behavior models, and thus support automation in testing.

In this section, we give a description of model-based testing structured into 6 steps, as illustrated in Figure 1.3. This structuring is inspired by [Utting 08], but with the extension to additionally include execution and verification of test cases.

1. *Creation* of the formal model by interpreting requirements or other informal descriptions of the intended functionality of the system. In this thesis we will assume the formal model to be an abstract model of a SUT.
2. *Validation* of the formal model, e.g., by animating the model.
3. *Generation* of a test suite from the model. This step is typically automated. Note that the generated test suite is on the same level of abstraction as the formal model, hence we use the term *abstract test suite*.
4. *Concretization* of each generated abstract test suite into a test suite that can be executed against the SUT.
5. *Execution* of test cases against a SUT in a controlled environment.
6. *Verification* of the execution of a test suite against the model. That is, abstractions of any output from the SUT are compared with the output for the corresponding generated test case, from the formal model. Further, any fault detected must be possible to trace back to the formal model. Note that a fault may be detected because of some errors remaining in the model.

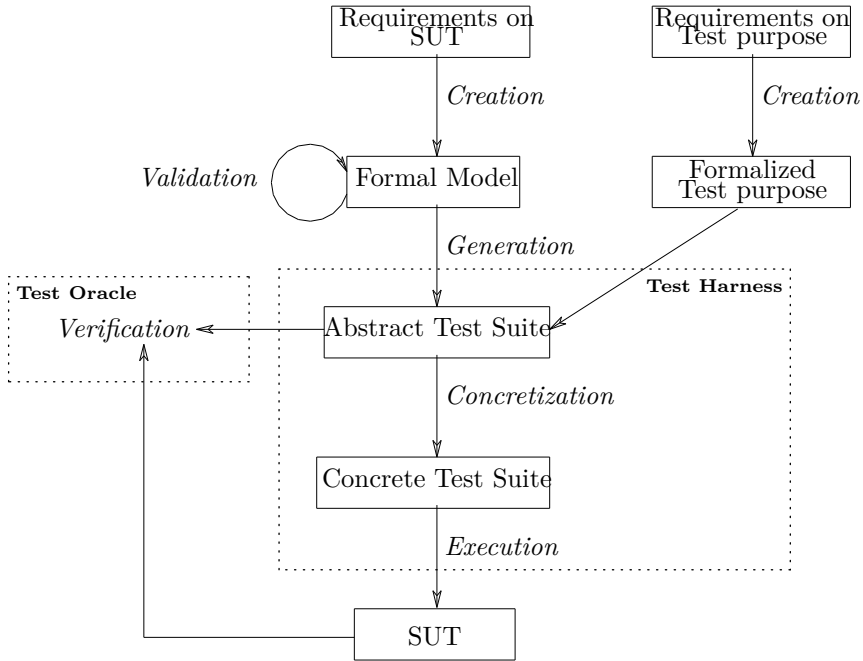


Figure 1.3. An overview of typical components in model-based testing.

In the following, we describe each of these steps in more detail.

1.5.1 Creation of a formal model

Model-based testing requires a formal model of the expected behavior of the SUT. Typically, this model represents an abstraction of the SUT. As illustrated in Figure 1.3, the model should be generated from the requirements, independently from the implementation. This is typically a manual process, but there exists tools e.g., Spec Explorer [Veanes 08, Sarma 10] to aid this process.

In this thesis we are mainly interested in the testing of communication protocols, see e.g., [Lai 02, Bishop 05] for overviews of model-based protocol testing. Models of protocol behavior often have the form of state machines. In this introduction, we illustrate state machines by an example.

Example 1.2 Figure 1.4 (from [Hong 02]) shows a state machine, which specifies the behavior of the controller of a simple coffee machine. The controller interacts with a user by the events *reset()*, *insert(X)*, *coffee()* and *show(M)*, and a brewer by means of events *make()* and *done()*. In order to brew a cup of coffee, a user first needs to pay for the coffee. A

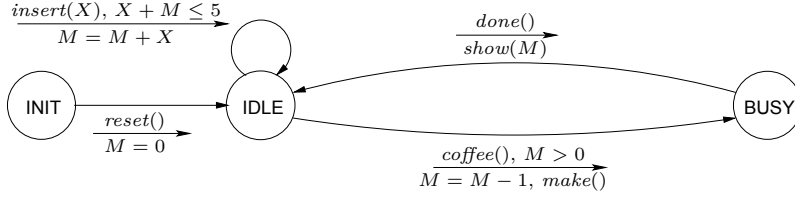


Figure 1.4. An abstract model specifying the controller of a simple coffee machine.

cup of coffee costs 1 unit, so whenever at least 1 unit is inserted and the coffee button pressed, brewing will start. When finished, the amount of money left is shown and the machine is ready to brew a new cup of coffee. Note that this model only specifies the behavior when a maximum of 5 units is inserted. The state machine has three locations: **INIT**, **IDLE** and **BUSY**, and one state variable M holding number of units inserted but not yet used. Upon the reception of an event, the machine may perform some actions and move to a new control state. For instance, in control state **IDLE**, when the event $insert(X)$ occurs, and $X + M$ not exceed 5, then the amount M of units left is increased by X , and the state machine remains in control state **IDLE**. \square

It is important to keep in mind that the formal model of the SUT is in general an *abstraction* of the desired behaviors of the SUT. For example, in Figure 1.4, the event $insert(X)$ is an abstraction of what actually happens in the SUT, namely that X coins, each being worth 1 unit, are inserted into the machine. Also, the actual representation of the variable M in the SUT may be more complex than just a simple number. Such abstractions make it possible to subject the model of different forms of analysis (by humans or computers), and allow the generation of large test suites. Different forms of abstractions for model-based testing are discussed in [Prenninger 05].

1.5.2 Validation of the formal model

The model should be validated, to ensure that the behavior it represents indeed satisfies intended requirements. Since state machine models are formal and executable, we can use animation, simulation, or formal verification. An example of a formal verification technique is model checking [Clarke 99], where a tool automatically examines whether all possible behaviors of a state machine model satisfy a given property. For example, for the model of the controller in Figure 1.4 it can be validated that the number of output events of form $make()$ always is less or equal to the number of inserted units for any possible test case.

1.5.3 Test Suite Generation

The validated formal model is the basis for generating a test suite. For example, from the model of the controller in Figure 1.4 it is possible to derive test cases such as

$$\begin{array}{ccccccc} \text{reset()}/ & \rightarrow & \text{insert(1)}/ & \rightarrow & \text{coffee()}/\text{make()} & \rightarrow & \text{done()}/\text{show()} \\ & & \rightarrow & & \text{insert(1)}/ & \rightarrow & \text{coffee()}/\text{make()} & \rightarrow & \text{done()}/\text{show()} \end{array}$$

This particular test case brews 2 cups of coffee where each cup of coffee is ordered immediately after inserting a coin. The model also describes many aspects of functionality that are not exercised by this test case, e.g., that several coins may be accumulated to allow ordering a sequence of coffee cups, or that coins may not be accepted before the *done()* input, that should ideally be verified by a test suite.

When a generated test suite grows larger, representation of test cases becomes a problem. It may then be more efficient with a symbolic representation. Thus, from a formal model of a SUT we generate symbolic test cases, each with a condition expressed over input parameters. Each symbolic test case can be seen as an equation where each instantiation of the input parameters also satisfying the condition represents an abstract test case. A corresponding symbolic test case to the above abstract test case can be written as

$$\left\langle \begin{array}{ccccccc} \text{reset()}/ & \rightarrow & \text{insert}(p_1)/ & \rightarrow & \text{coffee()}/\text{make()} & \rightarrow & \text{done()}/\text{show()} \\ & & \rightarrow & & \text{insert}(p_2)/ & \rightarrow & \text{coffee()}/\text{make()} & \rightarrow & \text{done()}/\text{show()} \end{array} \right\rangle ,$$

$$\left(\begin{array}{l} p_1 + 0 \leq 5 \\ \wedge \quad p_1 > 0 \\ \wedge \quad p_2 + 0 \leq 5 \\ \wedge \quad p_2 > 0 \end{array} \right) \rangle$$

A challenge with test suite generation is to generate a test suite that increase confidence in the correctness of the SUT (e.g., by exercise as much of the modeled behavior as possible) to a low cost (e.g., by generate a test suite with as few test cases as possible). In practice, this goal is difficult to achieve. There are different kinds of strategies for the generation of test suites, see e.g., [Anand 13]. In the following subsections, we describe some common strategies.

Partition Testing

In partition testing, the input domain is separated into several sub-domains and each test case is selected to ensure coverage of some sub-domain. For example, assume an input parameter to the SUT that assumes values in the integer domain. Using a partition criterion we can

then, for example, separate this domain into sub domains with negative and positive integers. Test cases can then be selected so that at least one test case use a positive integer and at least one test case a negative integer as a value of the input parameter.

Coverage Based Testing

In coverage based testing, test suites are generated by selecting test cases which cover some structural elements of the model, such as locations, edges, paths, and variables. The corresponding coverage criteria are then called *location coverage*, *edge coverage*, etc. This notion of coverage is reused from white-box testing. Most coverage criteria can be classified into *control flow oriented* coverage criteria and *data flow oriented* coverage criteria. Control flow oriented coverage criteria are based on logical expressions in the formal model which determine branches, and data flow oriented coverage criteria focus on the way values are associated to their variables and how these associations affect the execution of the test case [Vilkomir 01].

Random Testing

In random testing, test suites are constructed by randomly selecting test cases. The simplest form of random testing assumes a uniform probability distribution over all possible test inputs of the SUT. But test cases may also be biased in some direction, e.g., according to expected usage of the system. More directions on how random testing can be used effectively can be found in the case study by Ciupa et al. [Ciupa 07].

There is an ongoing, unresolved, debate on whether random testing can find faults faster than, e.g., partition or coverage based testing. The standard book by Beizer [Beizer 90] argues that faults often are found much faster by non-random methods. This has been questioned by e.g., [Ntafos 01], claiming that (1) partition testing to be more expensive than random testing, and that (2) the performance of random testing is rather close to partition testing. The main difference is that partition testing distributes test cases over partitions in an even manner, such that each partition is guaranteed at least one test case. Thus, if faults are distributed evenly over the model, then the advantage of partition testing is limited when the random test suite is larger than the number of partitions, and it decreases as the size of the test suite increases.

Combinatorial Testing

Combinatorial testing, [Mandl 85, Grindal 07], is a black-box technique which can be used when test cases can be characterized in terms of values of a set of parameters. Since for practical systems, it is intractable to execute tests for all combinations of parameter values, a subset of the possible combinations must be selected. In combinatorial testing, a

test suite is generated by combining parameter values according to some combination strategy.

Test Case:	TC_1	TC_2	TC_3	TC_4	TC_5	TC_6	TC_7	TC_8	TC_9
p_1	1	1	1	2	2	2	3	3	3
p_2	1	2	3	1	2	3	1	2	3
p_3	1	2	2	1	2	2	1	2	2

Table 1.1. *Test suite that aligns with a 2-wise combination strategy. for a system with 3 parameters with domains $p_1, p_2 \in \{1, 2, 3\}$ and $p_3 \in \{1, 2\}$.*

An example of a combination strategy is the *2-wise* combination strategy [Cohen 97] which requires that every possible pair of values of any two parameters is included in some test case. An example of a test suite generated from the 2-wise combination strategy can be found Table 1.1 where we in total have 3 parameters, p_1, p_2, p_3 , with domains $p_1, p_2 \in \{1, 2, 3\}$ and $p_3 \in \{1, 2\}$. It can be noted that the $3 \cdot 3 + 2 \cdot 3 + 2 \cdot 3 = 21$ possible pairs that need to be covered in this case can be handled by only 9 test cases.

Fault-based testing

In fault-based testing focus is not on a particular coverage criteria of a specification, but on faults that should be detected. Perhaps, the most well-known fault-based strategy is mutation testing see e.g., [Aichernig 09]. Although introduced for mutation of programs it has lately been adopted to model-based testing [Aichernig 12] where test cases covering a mutated formal specification are generated. Mutation based testing is based on the hypotheses that: (1) test cases that detect simple faults are likely to also detect complex faults and (2) SUT's are close to being correct. In mutation testing faults are introduced by syntactically change the specification into mutants. A test suite is then generated with test cases that would kill the mutants. Thus, when executing the generated test suite against a SUT, a fault is detected if the SUT behaves identically to the test case.

1.5.4 Concretization

As emphasized in Section 1.5.1, the formal model of the SUT is in general an *abstraction* of the desired behaviors of the SUT. This reduces the complexity involved when generating test suites automatically. However, it implies that the generated test cases are also abstractions, and cannot be executed directly on the actual SUT. For instance, test suites generated

from the model in Figure 1.4 will contain events of form *insert*(*X*), which cannot directly be provided as input to the SUT. Therefore, the generated *abstract test cases* must be concretized into *concrete test cases*, which can actually be executed. Occurrence of a *insert*(*X*) event in a test case could then cause a test environment to trigger a (real) insertion of *X* units into the implementation of the brewer.

1.5.5 Execution of test cases

Test cases in a test suite can be executed sequentially one by one or concurrently. Test suites can be generated from the formal model before execution of the test suite starts (off-line). Test suites may also be generated while executing the test suite (on-the-fly), see e.g., [Fernandez 96, Vries 00]. Thus, test suite generation and concretization of test cases are made during execution and verification of test cases. For large test suites this may be an advantage as test cases do not need to be stored, This comes with the cost of regenerating the test suite for each test execution. Note also that in order to execute test cases concurrently, also the on-the-fly generation must be made concurrent.

1.5.6 Verification

To ensure that an executed test case behaves as expected, i.e., according to the formal model, the test case must be verified. That is, we need to verify that any output data from the SUT does not conflict with the corresponding output data in the formal model,

As motivated in the preceding subsection, the concrete output data from the SUT is not at the same level of abstraction as the output data in the formal model. Thus, the output from the SUT must be abstracted in order to be compared with the predicted output in the abstract test case. Note that it is not a good idea to concretize the predicted output of the abstract test case and compare it with the concrete output of the SUT. The reason is that typically, each abstract output has many corresponding concrete outputs, whereas each concrete output has a unique abstraction.

1.6 Contributions of this Thesis

Despite the significant developments in model-based testing in the last decades [El-Far 02, BJK 04, Utting 07, Marinescu 15], its impact on industrial practice has still been rather limited. A number of commercially available tools exist, but still most testing in industry relies on manually constructed test cases.

The work in this thesis started from the need to systematically test a newly developed communication protocol application in a small software development company. We therefore started to develop automated tool support for model-based testing of communication protocols. A major goal was to make the tools suitable for industrial usage. This resulted in the implementation of **ERLY MARSH**, our own tool chain and work flow for model-based testing, based on the structure described in Figure 1.3, and resulting in a methodology described in Figure 1.5.

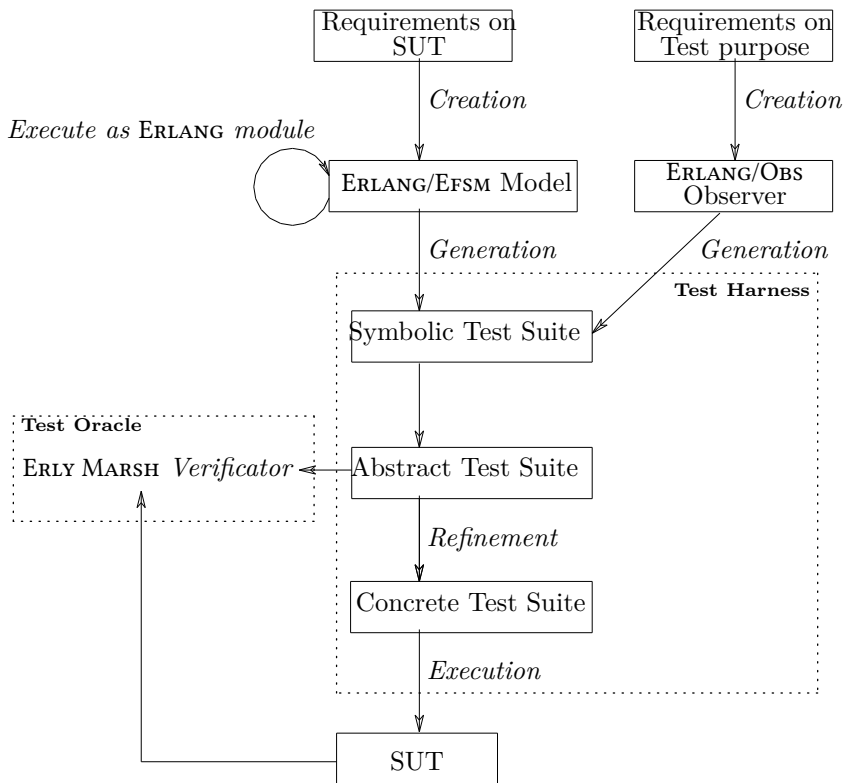


Figure 1.5. Components in model-based testing as used in this thesis.

For each step, we have examined how it can be best realized from an industrial perspective. On the way, we have produced a number of contributions to different phases of model-based testing. In addition, our tool chain represents a complete realization of this technology, and we have also evaluated it on a significant industrial case study.

In this section, we survey in more details our specific contributions to model-based testing.

1.6.1 Modeling Language

A difficulty for the adoption of model-based testing is the issue of how test engineers can construct test models effectively. Modeling is a new activity to many developers and testers, and the production and maintenance of high-quality models can be cost- and labor-intensive. Modeling can be made easier and more efficient by the availability of suitable modeling languages. One important criterion for a good specification language is that it should have just enough capabilities to express desired properties in a natural way. Another criterion is that it should be suitable for formal manipulation and analysis. Also, it must be easy to understand by all parties involved in the software development process. It seems reasonable that the choice of modeling language should depend on the type of software that is being tested.

Our work is intended for the testing of communication protocol software. The programming language `ERLANG` is frequently used for programming complex communication protocol software with a straightforward notation and the concept of sending and receiving messages is part of the language. `ERLANG` is therefore an excellent basis also for a modeling language. We have therefore developed the specification language `ERLANG/EFsm`. It is based on a subset of the functional programming language `ERLANG`, but adds constructs that are central for modeling control-intensive applications, such as communication protocols: control locations, state variables, configuration data, and structured messages.

State variables are distinct from normal `ERLANG` variables that obey the single assignment requirement. In order to develop automated test suite generation techniques and tools, `ERLANG/EFsm` is based on a subset of `ERLANG` obtained by restricting the expressiveness of data, and omitting exceptions, concurrency, and several other features of `ERLANG`.

Sharing syntax and semantics between specification and implementation language makes it easier to write specifications that can be understood and maintained by software developers who are familiar with the implementation language. Thus, for developers who are already familiar with programming in `ERLANG`, a modeling language based on `ERLANG` should be particularly easy to adopt.

1.6.2 Specifying Test Case Selection

In an industrial context, the desired size and thoroughness of a test suite can vary significantly, depending on the available testing budget, the cost of executing individual test cases, and the requirements on quality of the SUT. Coverage criteria is one mechanism for controlling the size and thoroughness of a test suite (see also Section 1.1). Most tools for model-based testing, e.g., [Tretmans 03, Friedman 02, Huima 07, Veanes 08] only allow

to generate test suites for a small set of coverage criteria. Since different coverage criteria are suitable in different situations, it is highly desirable that a test suite generation tool is able to generate test suites in a flexible manner, for a wide variety of different coverage criteria.

We have therefore developed a technique for specifying a large number of different coverage criteria, and a method for generating test cases according to such coverage criteria. The technique is based on the concept of *observers*. Observers can be thought of as a separate component in the model, which observes the execution of the state machine, and records which syntactical elements are “covered” by the execution. The advantage of the observer concept is its flexibility: it is easy to define a new coverage criterion, even criteria that have not appeared before in the literature. Observers are discussed in more detail in Section 3.

1.6.3 Efficient Generation of Test Suites

In all model-based testing it is essential that test suites are generated from a correct model. Examples include finding model inconsistencies, or to ensure coverage of certain requirements. If test suite generation is based on a formal model we have two options: (1) to formally validate that certain properties are satisfied by the model and (2) animate the model. Animation of a formal model written in `ERLANG/EFM` can be accomplished by generating an executable `ERLANG` module.

Given a model of the SUT in the form of a state machine, it is possible to automatically generate test cases from executions of this model, as outlined in Section 1.5.3. A naive approach to this problem would simply enumerate all the executions of this model. However, this approach quickly becomes impractical, due to the astronomical number of possible executions. This motivates use of symbolic execution to derive a more effective and user friendly symbolic representation of test cases. Symbolic representations describe not only single executions, but sets of them, and can increase the power of test suite generation significantly. Figure 1.6 illustrates the relationships between different representations of test cases, and the mappings between them that are performed during model-based testing.

Symbolic execution has been used in many ways for test suite generation, see Section 10.1. We have adapted this technique so that it can efficiently generate large test suites from specifications written in `ERLANG/EFM`. We have also incorporated the observer mechanism for defining coverage criteria and investigated different strategies for instantiating symbolic parameters.

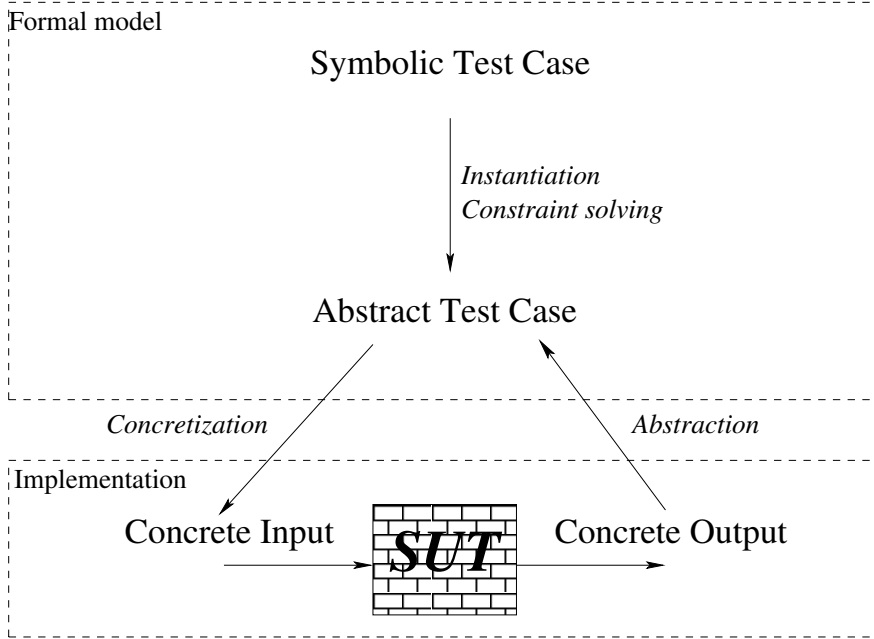


Figure 1.6. An outline of the abstraction levels used in this thesis.

1.6.4 Concretization of Generated Test Cases

Since an **ERLANG/EFM** model represents an abstract view of the behavior of SUT, we must concretize the generated abstract test suite to a format that corresponds to what the SUT expects and returns. In general the domains of the concrete values are much larger than the abstract domains and we must therefore carefully select concrete counterparts.

Concretization is handled by a SUT specific call-back module that can be configured with different concrete values on abstract parameter values. This makes it straightforward to use an abstracted generated test suite with different concrete values on a SUT. Further, versions of a SUT utilizing different interfaces can easily be tested by modifying the call-back module.

1.6.5 Efficient Test Execution and Verification

In order to execute a generated test suite, we must set up a test harness which interacts in the way each test case dictates. This is rather easy if the test execution is performed sequentially entirely within one computer. However, if test execution is performed concurrently or the test harness must be distributed over several different components in a physical network, mechanisms must be developed for coordinating these

different components, so that they agree on which test case is currently executed and on the timing of inputs. In the implementation of **ERLY MARSH**, we coordinate the different components by means of test case identifiers that are embedded within the messages exchanged with the SUT. This solution obviates the need for additional synchronization and communication between the different components of the test harness, and is rather simple to implement. We describe it in Section 6.5.

After executed test cases have been verified for correctness it is essential that the results are presented to a user with all relevant related information. Further, presentation of the results executing large test suites may be organized in several ways. Thus, report generation calls for the need of a database. For the results to be easier to understand, also other effects at the SUT that can be associated with the executed abstract test case can be essential. Examples of such effects can be e.g., logs, counters, performance measures etc. We describe it in Section 6.7.

1.6.6 Evaluation of Different Test Generation Strategies

Having developed a tool chain for model-based testing implementing the research contributions in Section 1.6.1-1.6.5 we further investigated how to generate test suites, such that we get the best trade off between costs and benefits? That is, an evaluation on whether model-based testing in general and the **ERLY MARSH** tool in particular is feasible in a real industrial size protocols and how it can be used most efficiently. We did this by creating a formal specification for an existing industrial protocol, from which test cases are automatically selected, executed and validated. We then evaluated selected techniques based on coverage based testing and random testing, and compared the results with an existing “manually” created test suite. In particular we study different strategies for: *Selecting* test cases for a test suite, to test as efficiently as possible (low cost) while gaining maximum coverage and fault detection (high benefits). We consider both the selection of symbolic test cases, and the selection of abstract test cases. *Executing* the set of selected test cases as efficiently as possible with respect to time and fault detection effectiveness.

1.7 Organization of Thesis

In Section 2-6 our contributions on methods and tools for test suite generation are presented. These sections covers test suite creation from a formal model and execution of generated test suites against an implementation. We focus on how to *specify* the SUT, with a suitable formal model from which test suites are generated by introducing a new specification language **ERLANG/EFSM** in Section 2, and *test* the SUT, with test cases generated from the formal model, and automatically validating the results. To control the selection of test cases we, in Section 3, define observers, including a specification language, **ERLANG/OBS**, for specification of observers. Section 4 give several examples on how coverage criteria can be specified using the observers. In Section 5 we use the observers, together with a symbolic approach, to select test cases to be included in a generated test suite. In Section 6 we present an overview of a tool, **ERLY MARSH**, for handling the generation and execution of test suites, given a model in **ERLANG/EFSM**.

Section 7-9 presents a case study where we evaluate our methods and tools. We use a commercially available industrial system that we generate test suites for and validate executions on. The foundation for this case study is found in Section 7. To study the effectiveness of the generated test suites we did some comparisons between different coverage based test case selections and random test case selection. The results are found in Section 8. To study the effectiveness of the tool we did some limited comparisons with other tools for testing **ERLANG** programs in Section 9.

Section 10 gives an overview of some related work, and Section 11 summarizes the results and experiences from this thesis and presents some ideas for future research.

2. A specification language based on **ERLANG**

This chapter introduces a specification language, which is primarily intended for automated test suite generation, and is based on the functional programming language **ERLANG**. We will refer to this specification language as **ERLANG/EFSM**.

This chapter is organized as follows. In Section 2.1, we motivate the introduction of a new specification language, and why we base it on **ERLANG**. Section 2.2 introduces **ERLANG/EFSM** by an example. Section 2.3 describes the syntax and intended meaning of the language constructs of **ERLANG/EFSM**, and Section 2.4 describes its operational semantics. Section 2.5 uses the semantics of Section 2.4 to show how each **ERLANG/EFSM** specification defines a state machines. Section 2.6 defines the concepts of runs and test cases. Section 2.7 describes a symbolic version of the operational semantics, intended for symbolic test case generation, and Section 2.8 shows how it corresponds to the operational semantics of Section 2.4. Section 2.9 defines symbolic analogues of the concepts of runs and test cases. Section 2.10 introduces an operation that normalizes the structure of a **ERLANG/EFSM** specification. Section 2.11 defines a transformation of **ERLANG/EFSM** specifications into executable form, which exploits the **ERLANG/OTP** `gen_fsm` behaviors.

2.1 Introducing **ERLANG/EFSM**

A difficult aspect in model-based verification and testing is the issue of how to produce specifications, and the cost of writing and maintaining them. An important factor is the choice of specification language. A good specification language should have just enough capabilities to express desired properties in a natural way and still be suitable for formal manipulation and analysis. Also, it must be easy to learn and understand by all parties involved in the software development process. Sharing syntax and semantics between specification and implementation language makes it easier to write specifications that are easy to understand and maintain by software developers who are familiar with the corresponding implementation language.

ERLANG is a functional language, and as such enables the user to write programs in a more succinct and abstract way than in many imperative

languages. It was designed in order to be suitable for programming communication software at a high level of abstraction, with features such as pattern matching, dynamic typing, and single assignment variables that allow to write compact programs for communication software.

ERLANG's ability to express communication behavior at a high level of abstraction also makes it suitable as the basis for a specification language. The use of ERLANG as a specification language was studied in [Jantsch 98] where ERLANG was found to be suitable for specification of large and complex control software, e.g., for telecom networks. For the automated generation of large test suites, protocol modules are very often represented as *state machines*, which have a number of control locations that are reached by sequences of message receptions and transmissions. Typically, state machines also have a number of state variables that are updated when moving from one control location to another. In order to support this style of specification, we have therefore developed the specification language ERLANG/EFSM. It retains ERLANG's ability to express communication behavior at a high level of abstraction, but also contains additional constructs for control locations, state variables, and structured messages. State variables of ERLANG/EFSM are distinct from normal ERLANG variables, which must obey the single assignment requirement. In order to support automated generation of test suites, only a restricted part of the ERLANG syntax is retained in ERLANG/EFSM. Most notably, we have restricted the expressiveness of data, as well as omitted exceptions, concurrency, and several other features of ERLANG.

We have also added two constructs in ERLANG/EFSM to support modularity. The first is that the assignment of attributes to objects need not be specified in an ERLANG/EFSM specification. For instance, the behavior of a protocol module may depend on certain attributes of the users involved in a communication session. We may then want to generate test cases for a range of such attributes, and supply specific assignments only when executing specific test cases, or when simulating the specification. We therefore include *configuration access functions* for assignments that need not be defined in an ERLANG/EFSM specification, but can be supplied externally when needed. The second is a facility for replacing complex expressions in a specification by simpler ones, by means of *abstraction macros*, inspired by a corresponding mechanism in ModEX [Holzmann 02] intended for the SPIN model checker [Holzmann 97]. This allows to suppress less important details in a specification in order to make test suite generation tractable. Typically, the replaced expression is used in an executable form of the specification, and the replacing expression is used when, e.g., generating test suites. Examples include standards when several values (e.g., status codes) are treated in the same way in the specification. Thus, where we have a clear understanding of the implementation, it

can be clearer to explicitly use concrete values in the specification instead of inventing new abstract parameters.

An alternative way to use `ERLANG` for specifying state machines would be to use the style provided by the `gen_fsm` behavior, which already exists in `ERLANG/OTP` [Erl 15]. We have chosen not to base our specification language on the `gen_fsm` behavior, since we want a specification language with a highlighted compact notation for control flow and state variables, which is not present in `gen_fsm`.

In order to leverage the available infrastructure in the existing `ERLANG` parser, and also to retain the style of `ERLANG` programs, the syntax for our extensions to `ERLANG` re-uses syntactical constructs that already exist in `ERLANG`. Thus, the syntax for transition clauses uses the same syntax as user defined functions, and configuration access functions use the syntax for function calls in `ERLANG`.

As a basis for subsequent formal manipulation of `ERLANG/EFsm` specifications, we develop a formal operational semantics for `ERLANG/EFsm` specifications. It is inspired by the semantics for `ERLANG` in [Fredlund 01], and is given as a big-step operational semantics, which is suitable for modeling transitions between control locations of a protocol module. To support symbolic techniques for test suite generation, we also develop a symbolic operational semantics where evaluated expressions may include symbolic parameters not evaluated to a value by the symbolic execution.

Since `ERLANG/EFsm` has extended `ERLANG` with constructs expressing state machines, an `ERLANG/EFsm` specification cannot be directly executed as an `ERLANG` program. To allow execution of `ERLANG/EFsm` specifications, we define a set of refactoring rules that transforms an `ERLANG/EFsm` specification into the form of a `gen_fsm` behavior, which is directly executable.

2.2 An Overview of `ERLANG/EFsm`

In this section, we introduce the overall structure of `ERLANG/EFsm` specifications, by means of an example. The overall structure of an `ERLANG/EFsm` specification is given in Figure 2.1. It consists of

- a declaration of control locations,
- a declaration of state variables,
- a declaration of configuration access functions, i.e., built-in functions to access configuration data during an execution,
- type declarations for state variables, configuration access functions, input expressions, output expressions, and user defined functions, which are optional, and
- for each control location, a list of *transition clauses* represent all edges in the state machine.

<code>-locations($\{l_0, stop, other\}$).</code>	Declarations of control locations
<code>-statevars($[v_1, \dots, v_n]$).</code>	Declarations of state variables
<code>-configs($[f_1/i_1, \dots, f/i_n]$).</code>	Declarations of configuration access functions
<code>-type($t_1()$, $tv_{11} \dots tv_{1i}$).</code> \vdots	Declarations of user defined types
<code>-type($t_n()$, $tv_{n1} \dots tv_{nj}$).</code>	
<code>-spec $f_1(p_{11}, \dots, p_{1i}) :: type_1$.</code> \vdots	Declarations of types for input/output events, user defined functions and configuration access functions
<code>-spec $f_n(p_{n1}, \dots, p_{nj}) :: type_n$.</code>	
$l(a_1, u_1, \dots, u_i)$ when $g_{11} \rightarrow ce_{11};$ \vdots	Transition clauses for all relevant input event types a_1, \dots, a_k , for each declared location l .
$l(a_1, u_1, \dots, u_i)$ when $g_{1m} \rightarrow ce_{1m}.$ \vdots	
$l(a_k, u_1, \dots, u_j)$ when $g_{k1} \rightarrow ce_{k1};$ \vdots	
$l(a_k, u_1, \dots, u_j)$ when $g_{kn} \rightarrow ce_{kn}.$	

Figure 2.1. An overview of a typical ERLANG/EFSM specification where i and j represent the arity of functions and events.

Before we explain the `ERLANG/EFsm` language in more detail, we illustrate by an example.

Example 2.1 The `ERLANG/EFsm` specification for the state machine in Figure 1.2 (from Example 1.1) is given in Figure 2.2. The locations in an `ERLANG/EFsm` specification are declared by

```
-locations({morning,
            [end_of_day],
            [workUU, workMA, preschool]}).
```

where `morning` is the initial location, `[end_of_day]` is a list with stop locations, i.e., locations from which there are no outgoing edges, and `[workUU, workMA, preschool]` is a list with all other locations that occur in the state machine.

The state variables `Progress`, `Stamina`, and `Day` are declared by declaration:

```
-statevars(['Progress', 'Stamina', 'Day']).
```

Note that the required use of quotes on declarations of variables is an artifact caused by our desire to be compatible with the `ERLANG` parser. Each day can be either a day when the kids should be collected from, or left at the preschool, by the author of the thesis. In the specification, we represent this by assigning the attribute `leave` or `collect` to each day. We want to leave it outside the specification what type of day any particular day is and consider it configurable (e.g., collectible from a calendar). Thus, to find out the attribute for a particular day we use the configuration access function `daytype` that takes a single argument. The `daytype` function is used in the `morning` location, where we have different configuration data for different values of the input event parameter `DayType`. The definition of this function can be considered as externally supplied, but must be declared

```
-configs([daytype/1]).
```

as a configuration access function with a single argument.

The domains of the state variables are declared by

```
-type('Progress'() :: 0..8 ).
-type('Stamina'() :: 0..2).
-type('Day'() :: any()).
```

and say that the value of `Progress` is any integer in the range $0, \dots, 8$, the value of `Stamina` is any integer in the range $0, \dots, 2$ and that `Day` can assume any value (i.e., it is untyped). The argument to the configuration access function `daytype` is declared by

```
-spec daytype(Day :: any()) -> collect | leave.
```

saying that it accepts an unspecified type on its single argument, and always returns either `collect` or `leave`. The input events are declared in a similar way by

```
-spec wakeup(TDay::any()) -> ok.  
-spec progress(X::0..2) -> ok.  
-spec incident(I::bug | pernilla_call |  
               kids_happy | kids_notok) -> ok.
```

and similarly, the output events are declared by

```
-spec checkout(Day::any()) -> ok.  
-spec checkin(Day::any()) -> ok.
```

where the declared return types have no significance for describing the state machine itself. However, auxiliary information can be provided for use by a tool, e.g., to aid presentation of sequences of input and output events. For example, our tool **ERLY MARSH**, is provided information on which external entity the test environment simulates, see Section 6.7.

For each declared location, except the stop location `end_of_day`, transition clauses specify the reaction to input events. The first argument of such a transition clause must match the input event type, and the remaining arguments must match the input event parameters. Thus, when in the `morning` location, at the occurrence of an input event `wakeup(monday)`, the transition clause matches, since the first argument matches the event type `wakeup` and the second argument `TDay` matches the event parameter `monday`. The transition clause specifies that first, the variable `Day` is bound to the value `monday`. Thereafter, depending on the value of `daytype(monday)`, one of the `if` clauses will be selected, i.e., either the output event `checkout(monday)` will be emitted and the location `workUU` will be entered (as specified by the tuple `{next_state,workUU}`), or the location `preschool` will be entered directly (as specified by the tuple `{next_state,preschool}`). □

```

-locations({morning,end_of_day,[workUU,workMA,preschool]}).
-statevars(['Progress','Stamina','Day']).
-configs([daytype/1]).
-type('Progress'() :: 0..8).
-type('Stamina'() :: 0..2).
-type('Day'() :: any()).
-spec daytype(Day::any()) -> collect | leave.
-spec wakeup(TDay::any()) -> ok.
-spec progress(X::1..3) -> ok.
-spec incident(I::bug | pernilla_call | kids_happy | kids_notok) -> ok.
-spec checkout(Day::any()) -> ok.
-spec checkin(Day::any()) -> ok.

morning(wakeup,TDay) ->
    Progress=0, Stamina=2, Day=TDay,
    if
        daytype(Day)==collect -> checkout(Day), {next_state,workUU};
        daytype(Day)==leave -> {next_state,preschool}
    end.

workUU(progress,X) ->
    Progress=Progress+X,
    if
        Progress>=8 -> checkin(Day), {next_state,end_of_day};
        true -> {next_state,workUU}
    end.

workUU(incident,I) ->
    Stamina=Stamina-1,
    checkin(Day),
    if
        Stamina<0;I==pernilla_call -> {next_state,end_of_day};
        I==bug -> {next_state,workMA};
        I==kids_notok -> {next_state,preschool}
    end.

workMA(progress,X) ->
    Progress=Progress+X,
    if
        Progress>=8 -> {next_state,end_of_day};
        true -> {next_state,workMA}
    end.

preschool(incident,I) ->
    Stamina=Stamina-1,
    if
        Stamina<0;I==kids_notok -> {next_state,end_of_day};
        I==kids_happy -> checkout(Day), {next_state,workUU}
    end.

```

Figure 2.2. ERLANG/EFSM representation of specification described in Example 1.1.

2.3 ERLANG/EFSM - an extension of ERLANG

In this section, we give a more detailed description of ERLANG/EFSM. We begin in Section 2.3.1 with the part of ERLANG/EFSM, that is also a subset of ERLANG, and in Section 2.3.2 we define the extensions in ERLANG/EFSM that are not found in ERLANG.

2.3.1 Syntax for a restricted set of ERLANG

In this subsection we describe the syntax of the part of ERLANG/EFSM, that is also a subset of ERLANG. For readers already familiar with ERLANG this section can therefore safely be ignored. Table 2.1 summarizes the ERLANG syntax base definitions, and Table 2.2, describes the included subset of ERLANG expressions. The ERLANG/EFSM syntax is given in standard Backus-Naur Form (BNF). In the definition of the syntax, terminals (e.g., 1, `true`, `and`) are written in teletype font. Meta variables, ranging over some syntactic domain, are written in italics with optional indices (e.g., *expr* and *value*₁). Repeated patterns are expressed with a dot notation, for example {*expr*₁, ..., *expr*_{*n*}} expresses a sequence of (ERLANG/EFSM) expressions separated by comma symbols and enclosed in braces. We sometimes also use \overline{expr} for *expr*₁, ..., *expr*_{*n*}, \overline{value} for *value*₁, ..., *value*_{*n*} etc. for sequences of length *n* where if *n* = 0 an empty sequence is implied. For common meta variables we also sometimes use a shorthand notation, *i* for *integer*, \overline{d} for \overline{value} and *e* for *expr* etc. A control character (*escape*) is, e.g., `\n` for the line feed character. A character (*char*) is any character defined by the ISO 8859-1 character set.

Basic values

The basic values consist of atoms, booleans, integers, the empty list, and the empty tuple. Atoms are values represented by a sequence of characters, e.g., `thesis` or `'Thesis'`. Atoms must be quoted if the first letter is *not* lowercase, otherwise atoms may or may not be quoted. Boolean is a subtype of atoms consisting of the atoms `true` and `false`. Integers consist of a sequence of digits.

Compound values

The compound values are the non-empty tuples ($\{d_1, \dots, d_n\}$) and conses ($[d_1|d_2]$) where *d*₁, ..., *d*_{*n*} are basic or compound values. A list is either the empty list [], or a cons [*d*₁|*d*₂] in which *d*₂ is itself a list. The syntax [*d*₁, ..., *d*_{*n*}] is a shorthand for the list [*d*₁|...|[*d*_{*n*}|[]]...].

Local variables

A local variable name starts with an upper case letter or underscore `_`, followed by digits, letters or underscores. A local variable can only be

<i>digit</i>	::=	0 ... 9	
<i>uppercase</i>	::=	A ... Z	
<i>lowercase</i>	::=	a ... z	
<i>digitletter</i>	::=	<i>digit</i> <i>uppercase</i> <i>lowercase</i> _	
<i>integer</i> (<i>i</i>)	::=	<i>digit</i> ⁺ - <i>digit</i> ⁺	
<i>unquotedatom</i>	::=	<i>lowercase digitletter</i> *	
<i>quotedatom</i>	::=	'(<i>char</i> <i>escape</i>) ⁺ '	
<i>atom</i> (<i>a</i>)	::=	<i>unquotedatom</i> <i>quotedatom</i>	
<i>bool</i>	::=	false true	
<i>string</i>	::=	[<i>char</i> ₁ , ..., <i>char</i> _{<i>n</i>}]	<i>n</i> ≥ 0
<i>u</i>	::=	<i>uppercase digitletter</i> * _	
<i>arithmetic</i>	::=	+ - * div	
<i>relop</i>	::=	=/ = == < > =< >=	
<i>logop</i>	::=	and not or xor	
<i>typetest</i>	::=	is_integer is_boolean is_record is_tuple is_list	
<i>guardop</i>	::=	<i>relop</i> <i>logop</i> <i>typetest</i> size	
<i>fun</i> (<i>f</i>)	::=	<i>atom</i>	
<i>type</i> (<i>t</i>)	::=	<i>atom</i>	
<i>bv</i>	::=	<i>atom</i> <i>bool</i> <i>integer</i> [] { }	
<i>value</i> (<i>d</i>)	::=	<i>bv</i> [<i>d</i> ₁ <i>d</i> ₂] { <i>d</i> ₁ , ..., <i>d</i> _{<i>n</i>} }	<i>n</i> > 0

Table 2.1. ERLANG *syntax base*.

ge	$::=$	$bv \mid u \mid v \mid ge\#a.a_k$	
		$\mid f(ge_1, \dots, ge_n)$	$n \geq 0$
		$\mid guardop(ge_1, \dots, ge_n)$	$n > 0$
$guard(g)$	$::=$	ge_1, \dots, ge_n	$n > 0$
cp	$::=$	$bv \mid u \mid [cp_1 \mid cp_2] \mid \{cp_1, \dots, cp_n\}$	$n > 0$
$expr(e)$	$::=$	$bv \mid u \mid v \mid [e_1 \mid e_2] \mid \{e_1, \dots, e_n\} \mid (e)$	$n > 0$
		$\mid \#a\{a_1 = e_1, \dots, a_n = e_n\}$	$n \geq 0$
		$\mid e\#a\{a_1 = e_1, \dots, a_n = e_n\}$	$n \geq 0$
		$\mid e\#a.a_k$	
		$\mid v = e \mid cp = e$	
		$\mid guardop(e_1, \dots, e_n)$	$n > 0$
		$\mid arithmetic(e_1, \dots, e_n)$	$n > 0$
		$\mid f(e_1, \dots, e_n)$	$n \geq 0$
		case e of	
		cp_1 when $g_1 \rightarrow ce_1$;	
		\vdots	$n > 0$
		cp_n when $g_n \rightarrow ce_n$	
		end	
		\mid if $g_1 \rightarrow ce_1; \dots; g_n \rightarrow ce_n$ end	$n > 0$
$clexpr(ce)$	$::=$	e_1, \dots, e_n	$n > 0$
		$f(cp_{11}, \dots, cp_{1m})$ when $g_1 \rightarrow ce_1$;	
$fundef$	$::=$	\vdots	$n > 0$
		$f(cp_{n1}, \dots, cp_{nm})$ when $g_n \rightarrow ce_n$.	$m \geq 0$
$inttype$	$::=$	$integer \mid integer_1..integer_2$	
$rectype$	$::=$	$\#a\{a_1::tv_1, \dots, a_n::tv_n\}$	$n \geq 0$
bit	$::=$	$atom() \mid integer() \mid boolean()$	
tv	$::=$	$bit \mid t() \mid any()$	
		$\mid atom \mid bool \mid inttype \mid rectype$	
$recdec$	$::=$	$-record(a, \{a_1, \dots, a_n\}).$	$n > 0$
$vtypedec$	$::=$	$-type(t()::tv_1 \mid \dots \mid tv_n).$	$n > 0$
$ftypedec$	$::=$	$-spec f(u_1::tv_1, \dots, u_n::tv_n) \rightarrow tv.$	$n > 0$

Table 2.2. A restricted set of the ERLANG syntax.

bound to a value once and has a scope within the enclosing function clause. Local variables named solely with an underscore are *anonymous* and is never bound to a value. ERLANG/EFMS introduces an additional class of variables, *state variables*, which are further explained in Section 2.3.2.

Built-in functions

A built-in function is a function that is included in the basic ERLANG language and should not be further defined (by a user defined function in ERLANG/EFMS). In ERLANG (and ERLANG/EFMS) terminology a number of built-in functions are considered to be *operators*. That is, the application of an operator to its arguments (operands) should be written in infix notation, e.g., `, A=2`.

Informally, the binary relational operators (*relop*) and the arithmetic operators (*arithmetic*) compare their arguments under varying relations. The logical operators (*logop*) operate on booleans and are strict (both arguments are always evaluated). The unary type tests (*typetest*) operate on all values and return a boolean.

We next continue with the subset of ERLANG expressions in Table 2.2.

Guard expressions

Guard expressions are boolean conditions and must not have any side effects. The outermost expression in a guard expression must be either the application of a guard operator (*guardop*), or a configuration access function (*f*), to guard expression arguments. A guard expression argument is either a basic value (*bv*), a bound local (*u*) or state (*v*) variable, an access to a record field (*e#a.a_k*), or the application of an outermost expression as above. Note that any occurring configuration access function must also be declared, see below, and that comma “,” is a shorthand notation for the logical operator *and*. See Section 2.3.2 for further information on configuration access functions.

Patterns

Patterns in ERLANG/EFMS (*cp*) have the same form as values but may also contain local variables. If the pattern contain local variables not previously bound to a value, these are also implicitly bound to values when matching the pattern against a value. For example, a value `{a,b}` matches a pattern `{A,B}` and binds *A* to *a* and *B* to *b*. But the value `{a,b}` will not match the pattern `{A,A}`, since *a* and *b* are distinct. The pattern match operator `=` is used in an expression *cp = e* where *cp* is a pattern that may contain local variables and *e* is an expression. If *cp* matches *e* any previously unbound variable in *cp* will be bound to a value. An

anonymous local variable `_` is considered a shorthand for a fresh variable (a local variable not occurring elsewhere in the pattern, or in the enclosing expression). For example, the expression

```
case test(2) of {_,_} -> true; _ -> false end
```

is a shorthand for the expression

```
case test(2) of {X1,X2} -> true; X3 -> false end
```

where the local variables `X1`, `X2`, and `X3` do not occur elsewhere in the specification.

Expressions

Expressions are tuples, records, lists, pattern matches, function calls, and `if` and `case` expressions. A tuple expression $\{e_1, \dots, e_m\}$ and `cons` expression $[e_1|e_2]$ is evaluated by evaluating the sub-expressions e_1 to e_m . A function call expression $f(e_1, \dots, e_m)$ is evaluated by evaluating the body of the function f with the results of the evaluation of the sub-expressions e_1 to e_m . The function f is the name of a built-in function, a user defined function, a configuration access function, or an output event type. All expressions are evaluated by evaluating any sub-expressions e_1 to e_m in left-to-right order. Below we elaborate on different forms of expressions.

Records

A record is a data structure for storing a fixed number of elements. It has named fields and is similar to a `struct` in C. In `ERLANG` (and `ERLANG/EFSM`) records are represented as tuples, see e.g., [Barklund 99], where the first element in the tuple is the name of the record. Record definitions are global and must be declared by

```
-record (a, {a1, ..., am}).
```

In the record definition, a is the name of the record and a_1, \dots, a_m the names of the fields in the record. Given a record definition for a , a record can be created with $\#a\{a_1 = e_1, \dots, a_{m_1} = e_{m_1}\}$. A created record can (later) be updated with $e\#a\{a_1 = e_1, \dots, a_{m_2} = e_{m_2}\}$ where e is an expression evaluating to a (previously) created record. Note that we may choose to only create selected fields, leaving the remaining fields with a value `undefined`. Similar we may choose to only update selected fields, leaving the remaining fields unchanged. The current value of a field a_k can be accessed by $e\#a.a_k$ from a record expression e with name a .

User defined functions

A user defined function is defined with the syntax

$$\begin{aligned} &f(cp_{11}, \dots, cp_{1m}) \text{ when } g_1 \rightarrow ce_1; \\ &\quad \vdots \\ &f(cp_{n1}, \dots, cp_{nm}) \text{ when } g_n \rightarrow ce_n. \end{aligned}$$

i.e., as a sequence of function clauses separated by semicolons and terminated by a period. When the function is called, the first clause for which the formal parameter patterns $(cp_{k1}, \dots, cp_{km})$ match the actual arguments, and the guard expression (g_k) evaluates to **true**, is evaluated.

Case and if expressions

The **case** expression

```
case e of
  cp1 when g1 -> ce1;
  ⋮
  cpn when gn -> cen
end
```

is evaluated by first evaluating e to a value d , then matching d against patterns cp_1, \dots, cp_n , and checking whether the corresponding guard expression g_i evaluates to true. For a clause with index i that has both a matching pattern and a guard that evaluates to true, the clause expression ce_i is evaluated. If several clauses match, the first one (in left-to-right order) is chosen.

The **if** expression

```
if g1 -> ce1; ⋯ ; gn -> cen end
```

is a sequential choice construct from **ERLANG** such that the evaluation proceeds with the first clause expression ce_i for which the corresponding guard expression g_i evaluates to **true**. An **if** expression, such as the above, is equivalent to a **case** expression

```
case true of
  _ when g1 -> ce1;
  ⋮
  _ when gn -> cen
end
```

Type declarations

ERLANG is a dynamically typed language, therefore explicit type declarations for functions and local variables are optional. The same is also true for ERLANG/EFSM. User defined types are declared by

```
-type (t() :: tv1 | ... | tvn).
```

where t is a user defined type, previously not defined, with possible subtypes tv_1, \dots, tv_n . A sub-type tv is a built-in type (*bit*), a user defined type ($t()$), the set of all possible terms (*any()*), a compound value (*d*), a range of integers (*integer..integer*), or a record field type declaration $\#a\{a_1::tv_1, \dots, a_n::tv_n\}$.

Types for arguments and return values in functions (a specification or contract in ERLANG terminology) can be further declared by

```
-spec f(u1::tv1, ..., un::tvn) -> tvout.
```

where f is the name of a function with arity n , and each u_i is the name of argument i , declared to be of type tv_i . The function f returns a value with type tv_{out} .

Example 2.2 A user defined type, `daystatus()`, that contains only the atoms `collect` and `leave` can be declared by

```
-type(daystatus() :: collect | leave).
```

The types for arguments and return values, for a configuration access function `daytype` is declared by

```
-spec daytype(Day :: any()) -> daystatus().
```

where we do not restrict the value of the only argument (`Day`) but declares its type to be `any()`. The function returns either of the values in the previously defined type `daystatus()`. \square

Note that in test suite generation, when representing conditions (see Section 6.4), the domains size may be required to be known. Thus, when not explicitly given, a domain size of 2 will be assumed.

2.3.2 ERLANG/EFSM extensions to ERLANG

ERLANG/EFSM is specifically targeted towards creating specifications for efficient test suite generation. Therefore, ERLANG/EFSM extends the subset of ERLANG described in the subsection 2.3.1 by constructs that are found in Table 2.3.

v	$::=$	$uppercase\ digitletter^*$	
$atlist$	$::=$	$[atom_1, \dots, atom_n]$	$n \geq 0$
$locdec$	$::=$	$-locations(\{atom, atlist_1, atlist_2\}).$	
$statevardec$	$::=$	$-statevars([v_1, \dots, v_n]).$	
$configdec$	$::=$	$-configs([f_1/integer_1, \dots, f_n/integer_n]).$	$n \geq 0$
$abstrscope$	$::=$	$all \mid \{integer_1, integer_2\} \mid \{f, integer\}$	
$abstrdec$	$::=$	$-abstr(\{abstrscope, string_1, string_2\}).$	
$specdec$	$::=$	$recdec \mid vtypedec \mid ftypedec$ $\mid statevardec \mid configdec \mid abstrdec$	
$declarations$	$::=$	$locdec\ specdec^*$	
$spec$	$::=$	$declarations\ fundef^+$	

Table 2.3. ERLANG/EFSM *syntax extensions to ERLANG.*

Locations

The locations that occur in a specification must be declared by

$-locations(\{startloc, stoplocs, otherlocs\}).$

where *startloc* is the initial location, *stoplocs* is a non-empty list with the stop locations, i.e., locations from which there are no outgoing edges, and *otherlocs* is a list with all locations that are neither initial nor a stop location. We use *L* to denote all locations in the ERLANG/EFSM specification.

State variables

In ERLANG/EFSM we are specifically interested in specification of EFSMs that have state variables. For this purpose, an additional class of variables, *state variables* have been introduced to ERLANG/EFSM. State variables share the syntax with local variables in Table 2.1. We will use *u* when referring to a local variable and use *v* when referring to a state variable. State variables differ from local variables in that they must be explicitly declared, have a global scope, cannot occur in patterns, and are not restricted to single assignment. There is also no concept of anonymous state variables. State variables are declared by

$-statevars(quotedvarlist).$

where *quotedvarlist* is a list of quoted state variables. State variables are quoted only to keep compliance with the standard ERLANG parser; they

are declared with quotes, but used without quotes. For example, a state variable `Progress` is declared by quotes as

```
-statevars(['Progress']).
```

but occurs in a guard expression without the quotes as `Progress >= 8`. The (optional) type for a state variable is declared using the same syntax as for a user defined type, see Section 2.3.1, where the user defined type (t) shares the name with the (quoted) name of the declared state variable. For example, the type for `Progress` is declared by

```
-type('Progress'() :: 0..9 ).
```

i.e., `Progress` can take any integer value in the range $0, \dots, 9$.

Before usage, a state variable must be bound to a value. A state variable is bound to a value with an assignment $v = e$ where v is a state variable and e is an expression.

Configuration access functions

Configuration access functions access configuration data, e.g., assignment of attributes to certain objects, during an execution. For example, in Example 2.1 we used `daytype/1` to access the type of a particular day, thereby setting up a particular configuration. During an execution it is possible we may want to find out the configuration, e.g., the type of day for several days. For such occasions, an argument can be passed to the configuration access function. This allow us to parameterize configuration data depending on e.g., some input event parameter. Configuration access functions are declared by

```
-configs([f1/integer1, ..., fn/integern]).
```

where $n > 0$ and each f_i is the name of a configuration access function and $integer_i$ its arity. An (optional) type declaration for a configuration access function has the same syntax as for user defined functions, see Section 2.3.1.

Configuration access functions are handled by the `ERLANG/EFsm` parser as built-in functions that can be supplied externally and are not explicitly defined in an `ERLANG/EFsm` specification. Exactly how a definition is given depends on the tool used. Using our tool, `ERLY MARSH`, definitions to evaluate a configuration access function are given by a call-back module. During symbolic execution, described in Section 2.7, configuration access functions are treated as symbolic parameters, and need not be evaluated.

Input expressions

Input expressions are evaluated to input events, which may trigger transitions of the state machine. An input expression is of form $f(e_1, \dots, e_m)$

where f is the input event type and e_1, \dots, e_m are input expression parameters. In an **ERLANG/EFSM** specification, input expressions are pattern matched against the formal parameters in the transition clause, see below. Types for input expression parameters are declared by the same syntax as for user defined functions, see Section 2.3.1.

Output expressions

Output expressions evaluate to events emitted from the SUT, and have the syntax $f(e_1, \dots, e_m)$ where f is the output event type and e_1, \dots, e_m are output expression parameters. The output event type, f , must not conflict with the name of any built-in function, user defined function, or configuration access function. Output expressions are not explicitly declared and thus any expressions not explicitly declared (or part of the **ERLANG/EFSM** syntax in this section) is considered as an output expression. The (optional) types for output expression parameters are declared by the same syntax as user defined functions, see Section 2.3.1. Output expressions always evaluate to **true**.

Transition clauses

Each location in a state machine may be connected with another location via an edge. In **ERLANG/EFSM** we use *transition clauses* as a compact representation of edges. Each transition clause represent a set of edges that may be triggered on the occurrence of an input event with some arity m . A set of transition clauses is of form

$$\begin{aligned} l(a_0, u_1, \dots, u_m) \text{ when } g_1 \rightarrow ce_1; \\ \vdots \\ l(a_0, u_1, \dots, u_m) \text{ when } g_n \rightarrow ce_n. \end{aligned}$$

where l is the location from which the edges originate, and for each transition clause with index j , for $j = 1, \dots, n$,

- a_0 is the input event type,
- u_1, \dots, u_m are local variables and represent formal parameters to an input expression with arity m ,
- g_j is a guard expression, and
- ce_j is a clause expression that must return a tuple of form

$$\{\text{next_state}, l'\}$$

which represents a transition to the next location l' (which may be l).

The transition clauses originating from a location are defined with the same syntax as user defined functions, but with restrictions on patterns allowed in the function head. The location l must be declared as a location, and a_0 is an atom and u_1, \dots, u_m are local variables. Any local

or state variable that occurs in the guard expression (g_j) or in the clause expression (ce_j) of a transition clause must be bound to a value before its use.

In an **ERLANG/EFSM** specification there exists a set of transition clauses for each relevant (i.e., that can occur) combination of location, input event type and arity of input event type. In particular, note that in contrast to **ERLANG**, there can exist several transition clauses each with different input event type a , but with the same arity m and location l .

The translation of a transition clause in **ERLANG/EFSM** into edges (represented by edge clauses in a symbolic state machine, see Section 2.9), can be thought of as unfolding a branch in a tree ce_j until a return tuple, i.e., $\{\text{next_state}, l'\}$, is reached and we get a possible execution between the locations l and l' . The translation of transition clauses in a state machine is further elaborated on in Section 2.10.

Abstraction macros

A specification may have multiple purposes and it is therefore desirable to easily be able to rewrite the original specification. For this reason, we introduce an *abstraction macro* preprocessor extension defined as rewriting rules between valid **ERLANG** expressions and valid **ERLANG/EFSM** expressions. Abstraction macros can be used in several ways, including the following.

1. Variables with large domains can be abstracted into corresponding variables with smaller domains. For example, an input event parameter A with a large integer domain and an expression $A \geq 10$ in a guard expression would generate different abstract test cases for all values of A greater than 10. With no other constraints on the guard expression and a large domain of A , this number can be significant. If the actual value of A is not relevant, the expression $A \geq 10$ can be replaced by a boolean variable. Thus, the number of possible abstract test cases resulting from $A \geq 10$ has been reduced to two.
2. Parts of the specification that are not considered relevant for test suite generation can be hidden. For example, a specification may reflect how a SUT reacts when input event parameters have unexpected values. If it can be expected the behavior of the SUT is similar regardless of where in a test case such an input event parameter occurs, we may want to omit this from test suite generation and test separately. Thus, the size of the generated test suite can be limited.
3. A specification not written in **ERLANG/EFSM**, can be translated to a **ERLANG/EFSM** specification, from which we can generate test suites. For example, an **ERLANG** expression

```
LogServerPid!{max,A,is,Result}
```

for passing a message to `LogServerPid` is not a valid `ERLANG/EFSM` expression and needs to be replaced before test suite generation. Abstraction macros are declared by

```
-abstr(abstrscope, string1, string2).
```

where *string*₁ is a text string that may contain any characters, and *string*₂ is a string with a valid `ERLANG/EFSM` expression. The scope of the abstraction macro is controlled by *abstrscope* and can be:

- `all`, i.e., the entire specification.
- A tuple `{integer1, integer2}`, denoting a range of lines from line *integer*₁ to line *integer*₂.
- A tuple `{f, integer}`, where *f* is a user defined function definition with arity *integer*, or a location in a transition clause matching input events with arity *integer* – 1

Abstraction macros are applied on type declarations and transition clauses in the order they appear, within the associated scopes.

If *string*₁ is an `ERLANG` expression, white space and order of commutative subexpressions is not significant. For example,

```
-abstr(all, "is_integer(A), A<10", "Abool==true").
```

is equivalent with

```
-abstr(all, "A<10, is_integer(A)", "Abool==true").
```

Example 2.3 The following function clause from an `ERLANG` implementation of a max operation returns the maximum value of `A` and `B` if they are both integers, otherwise it returns `undefined_value`. In addition it also logs the result on a log server by sending a message to the log server process.

```
max10(A, LogServerPid) ->
    Result = if
        is_integer(A) ->
            if
                A < 10 ->
                    true;
                A >= 10 ->
                    false;
            end;
        not is_integer(A) ->
            undefined_value
    end,
    LogServerPid!{max, A, is, Result},
    Result.
```

We want to create abstractions in a way so that an abstraction of the above user defined function clause can be used in an `ERLANG/EFsm` specification. Note that message passing using the `ERLANG` operator `!` is not supported in `ERLANG/EFsm`. We therefore create an abstraction macro to remove it.

```
-abstr(all,"LogServerPid!{max,A,is,Result}","",").
```

Second, we are only interested in generating test cases when `A` is known to be an integer. Thus we create an abstraction macro for the integer testing guard expression

```
-abstr(all,"is_integer(A)","true").
```

to project the model to cases where the guard expression `is_integer(A)` is evaluated to `true`. Third, we choose to create abstraction macros for the remaining guard expressions `A>=10` and `A<10`, and replace the expressions with tests of a new boolean local variable `Abool`.

```
-abstr(all,"A<10","Abool==true").
```

```
-abstr(all,"A>=10","Abool==false").
```

Finally, as we have abstracted away all usage of the `A,LogServerPid` arguments these can be removed and replaced with `Abool` in the function head

```
-abstr(all,"max10(A,LogServerPid)","max10(Abool)").
```

After the above abstraction macros have been applied the resulting `ERLANG/EFsm` expression is:

```
max10(Abool) ->
  Result=if
    true ->
      if
        Abool==true ->
          true
        Abool==false ->
          false;
      end,
    not true ->
      undefined_value
  end,
  Result.
```

As we created an abstraction for the head of the user defined function, additional abstraction macros are required to also abstract any usage of this function. In particular, this may require introduction of new variables in transition clauses where this function is used. □

2.4 Operational semantics of ERLANG/EFM

The semantics for ERLANG/EFM is given as a big-step structural operational semantics [Plotkin 81] in the form of transitions between structural states. In this thesis we use the term structural state instead of, the more standard, configuration, to avoid confusion with handling of configuration data. Each transition describes how an ERLANG/EFM expression in one structural state is evaluated to a value in another structural state.

Let A_a be the domain of input event types, A_b the domain of output event types, C the domain of names of configuration access functions, used to access constant configuration data, and D the domain of all Erlang values.

Definition 2.4 A *structural state* is a tuple

$$\langle e, \bar{b}, \sigma, \rho \rangle$$

where

- e is an ERLANG/EFM expression,
- \bar{b} is a sequence of output events, each of which is of form $b(d_1, \dots, d_n)$, such that $b \in A_b$ is an output event type and each $d_j \in D$; for brevity, we sometimes use b to denote an output event, not just the output event type,
- $\sigma \in V \rightarrow D$ is a global environment with bindings to the global state variables V , and
- $\rho \in U \rightarrow D$ is a local environment with bindings to the local variables U .

An empty (local or global) environment is denoted \bullet and whenever \bar{b} is the empty sequence we write ϵ . \square

The evaluation of an ERLANG/EFM expression depends not only on the expression and the current global and local environments, but also on the actual meanings of the configuration access function names in C . Therefore, define a *configuration environment* to be a mapping Δ from C to functions of the appropriate types. A *transition* between two structural states in a particular configuration environment is a triple, written as

$$\Delta \models \langle e, \sigma, \rho \rangle \Rightarrow \langle d, \bar{b}, \sigma', \rho' \rangle,$$

denoting that e can be evaluated in a global environment σ , local environment ρ , and configuration environment Δ , to a value d . We always assume the transition to start in a structural state with an empty sequence of output events, and thus omit it from the starting structural state. After the evaluation, \bar{b} is the sequence of generated output events, σ' is the new global environment with bindings of state variables, and ρ' is the new local environment with bindings of local variables.

The configuration environment Δ is unchanged throughout the transition, and also throughout sequences of transitions. In the following, we will therefore mostly omit it, e.g., in descriptions of transition rules.

As a shorthand notation we let $\langle \sigma, \rho \rangle \vdash g$ denote that

$$\langle g, \sigma, \rho \rangle \Rightarrow \langle \text{true}, \epsilon, \sigma, \rho \rangle,$$

i.e., that the guard expression g evaluates to **true** in a global environment σ and local environment ρ . Note that σ and ρ are not affected since guard expressions must not have side effects. Similarly, $\langle \sigma, \rho \rangle \not\vdash g$ is a short hand notation for when g evaluates to **false**.

The set of possible transitions is defined by the set of transition rules. A *transition rule* denotes the condition under which $\langle e, \sigma, \rho \rangle \Rightarrow \langle d, \bar{b}, \sigma', \rho' \rangle$ exists and is of form:

$$\frac{cond_1, \dots, cond_m \quad trans_1, \dots, trans_n}{\langle e, \sigma, \rho \rangle \Rightarrow \langle d, \bar{b}, \sigma', \rho' \rangle}$$

where $cond_1, \dots, cond_m$ are conditions under which the transition rule can be applied. This rule says that a transition rule is *enabled* for an ERLANG/EFM expression e when the conditions $cond_1, \dots, cond_m$ are satisfied and the transitions $trans_1, \dots, trans_n$ exist. We will omit the configuration environment in the presentation of rules, since it is always the same for all involved transitions.

2.4.1 Pattern matching

In ERLANG/EFM (and ERLANG), pattern matching is used to enable expressions to be evaluated and to bind values to local variables. We define a relation *match* (cp, d, ρ, ρ') in Figure 2.3 to denote a successful pattern match between a pattern cp and a value d within a local environment ρ . The resulting local environment ρ' is ρ updated with all bindings created when successfully pattern matching cp and d . The match relation is undefined for all cp not matching d . Note that state variables can not occur in patterns. Thus, a pattern match can not bind values to state variables.

2.4.2 Transition Rules for ERLANG/EFM Expressions

Basic and Compound values

All expressions in ERLANG/EFM can be evaluated to basic or compound values. In this section we give the transition rules for how this is accomplished. Basic and compound values are expressions and naturally

d is a basic or compound value	$match(d, d, \rho, \rho)$
u is an anonymous local variable	$match(u, d, \rho, \rho)$
u is a local variable, defined in ρ	$match(u, \rho(u), \rho, \rho)$
u is a local variable, not defined in ρ	$match(u, d, \rho, \rho[u \mapsto d])$
$match(cp_1, d_1, \rho_0, \rho_1) \quad \dots \quad match(cp_n, d_n, \rho_{n-1}, \rho_n)$	$match(\{cp_1, \dots, cp_n\}, \{d_1, \dots, d_n\}, \rho_0, \rho_n)$
$match(cp_1, d_1, \rho_0, \rho_1) \quad match(cp_2, d_2, \rho_1, \rho_2)$	$match([cp_1 cp_2], [d_1 d_2], \rho_0, \rho_2)$
$match(cp_1, d_1, \rho_0, \rho_1) \quad \dots \quad match(cp_n, d_n, \rho_{n-1}, \rho_n)$	$match(\overline{cp}, \overline{d}, \rho_0, \rho_n)$

Figure 2.3. Definition of the $match(cp, d, \rho, \rho')$ relation for a pattern match between a pattern cp and a value d . Initially we have a local environment ρ . After a successful pattern match the local environment ρ' holds ρ and all additional bindings of local variables, required for matching.

evaluate to themselves.

$$\frac{d \text{ is a basic or compound value}}{\langle d, \sigma, \rho \rangle \Rightarrow \langle d, \epsilon, \sigma, \rho \rangle}$$

Non-empty tuples and conses are evaluated by first evaluating all sub-expressions in a left-to-right order.

$$\frac{\langle e_1, \sigma_0, \rho_0 \rangle \Rightarrow \langle d_1, \bar{b}_1, \sigma_1, \rho_1 \rangle \quad \dots \quad \langle e_n, \sigma_{n-1}, \rho_{n-1} \rangle \Rightarrow \langle d_n, \bar{b}_n, \sigma_n, \rho_n \rangle}{\langle \{e_1, \dots, e_n\}, \sigma_0, \rho_0 \rangle \Rightarrow \langle [\{d_1, \dots, d_n\}, \bar{b}_1 \dots \bar{b}_n], \sigma_n, \rho_n \rangle}$$

$$\frac{\langle e_1, \sigma_0, \rho_0 \rangle \Rightarrow \langle d_1, \bar{b}_1, \sigma_1, \rho_1 \rangle \quad \langle e_2, \sigma_1, \rho_1 \rangle \Rightarrow \langle d_2, \bar{b}_2, \sigma_2, \rho_2 \rangle}{\langle [e_1|e_2], \sigma_0, \rho_0 \rangle \Rightarrow \langle [d_1|d_2], \bar{b}_1 \bar{b}_2, \sigma_2, \rho_2 \rangle}$$

In **ERLANG** (and **ERLANG/EFMS**) records are represented by tuples. We will therefore in this section omit to give explicit rules for record expressions.

Local and State variables

Local variables are bound to values by pattern matching. Local variables must not previously be defined in the local environment ρ since **ERLANG/EFMS** is single assignment.

$$\frac{\langle e, \sigma, \rho \rangle \Rightarrow \langle d, \bar{b}, \sigma', \rho'' \rangle \quad \text{match}(cp, d, \rho'', \rho')}{\langle cp = e, \sigma, \rho \rangle \Rightarrow \langle d, \bar{b}, \sigma', \rho' \rangle}$$

State variables are distinguished from local variables and are bound to a value by first evaluating an expression and then the global environment σ is updated with the result.

$$\frac{v \in V \quad \langle e, \sigma, \rho \rangle \Rightarrow \langle d, \bar{b}, \sigma', \rho' \rangle}{\langle v = e, \sigma, \rho \rangle \Rightarrow \langle d, \bar{b}, \sigma'[v \mapsto d], \rho' \rangle}$$

All variables must be bound to a value before they are used such that the value of a state variable is found in the global environment, σ , and the value of a local variable is found in the local environment, ρ . A local variable has precedence over a global variable with the same name.

$$\frac{v \text{ is defined in } \sigma}{\langle v, \sigma, \rho \rangle \Rightarrow \langle \sigma(v), \epsilon, \sigma, \rho \rangle} \quad \frac{u \text{ is defined in } \rho}{\langle v, \sigma, \rho \rangle \Rightarrow \langle \rho(u), \epsilon, \sigma, \rho \rangle}$$

Built-in functions

All arithmetic expressions, type declarations and relational and logical operators are handled in the same way. For example, the transition rule

for the logical operator disjunction is

$$\frac{\begin{array}{c} \langle e_1, \sigma_0, \rho_0 \rangle \Rightarrow \langle \text{bool}_1, \overline{b}_1, \sigma_1, \rho_1 \rangle \\ \vdots \\ \langle e_n, \sigma_{n-1}, \rho_{n-1} \rangle \Rightarrow \langle \text{bool}_n, \overline{b}_n, \sigma_n, \rho_n \rangle \end{array}}{\langle \text{ or } (e_1, \dots, e_n), \sigma_0, \rho_0 \rangle \Rightarrow \langle \text{ or } (\text{bool}_1, \dots, \text{bool}_n), \overline{b}_1 \dots \overline{b}_n, \sigma_n, \rho_n \rangle}$$

Guard expressions

Guard expressions compute boolean conditions and occur in case expressions, if expressions, and function definitions.

The transition rules for guard expressions are similar as for disjunction above, but do not update the local or global environment nor can output expressions occur.

Sequence Expressions

A sequence expression simply evaluates to the value of the last clause expression in the sequence. The value of the first clause expression is simply ignored. To avoid ambiguity and improve readability, the **ERLANG/EFM** expression is sometimes written with parenthesis in the following transition rules.

$$\frac{\langle ce_0, \sigma, \rho \rangle \Rightarrow \langle d, \overline{b}_0, \sigma'', \rho'' \rangle \quad \langle ce_1, \sigma'', \rho'' \rangle \Rightarrow \langle d', \overline{b}_1, \sigma', \rho' \rangle}{\langle (ce_0, ce_1), \sigma, \rho \rangle \Rightarrow \langle d', \overline{b}_0 \overline{b}_1, \sigma', \rho' \rangle}$$

If Expressions

An if expression is a sequential choice construct such that the evaluation proceeds with the first clause expression ce_i for which the corresponding guard expression g_i evaluates to **true**.

$$\frac{\begin{array}{c} \forall j, 1 \leq j < i : \langle \sigma, \rho \rangle \not\models g_j \\ \langle \sigma, \rho \rangle \models g_i \\ \langle ce_i, \sigma, \rho \rangle \Rightarrow \langle d, \overline{b}, \sigma', \rho' \rangle \end{array}}{\left\langle \left(\begin{array}{c} \text{if} \\ g_1 \rightarrow ce_1; \\ \vdots \\ g_n \rightarrow ce_n \\ \text{end} \end{array} \right), \sigma, \rho \right\rangle \Rightarrow \langle d, \overline{b}, \sigma', \rho' \rangle}$$

Case Expressions

The case expression is somewhat complicated to handle, since the choice of clause depends on both that a pattern matches and that a guard evaluates

to **true**. First, an expression e is evaluated to a value d . Sequentially, d is then matched against pattern cp_j , and if the match succeeds the guard expression g_j is evaluated in the local environment ρ_t . For the first clause i with a pattern, cp_i , matching d and guard expression g_i evaluating to **true** in the local environment ρ''' , the clause expression ce_i is evaluated.

$$\begin{array}{c}
\langle e, \sigma, \rho \rangle \Rightarrow \langle d, \overline{b_1}, \sigma'', \rho'' \rangle \\
\forall j, 1 \leq j < i : \text{match}(cp_j, d, \rho'', \rho_t) \text{ implies } \langle \sigma'', \rho_t \rangle \not\vdash g_j \\
\text{match}(cp_i, d, \rho'', \rho''') \\
\langle \sigma'', \rho''' \rangle \vdash g_i \\
\langle ce_i, \sigma'', \rho''' \rangle \Rightarrow \langle d, \overline{b_2}, \sigma', \rho' \rangle \\
\hline
\left\langle \left(\begin{array}{l} \text{case } e \text{ of} \\ \quad cp_1 \text{ when } g_1 \rightarrow ce_1; \\ \quad \vdots \\ \quad cp_n \text{ when } g_n \rightarrow ce_n \\ \text{end} \end{array} \right), \sigma, \rho \right\rangle \Rightarrow \langle d, \overline{b_1 b_2}, \sigma', \rho' \rangle
\end{array}$$

User Defined Functions

A call to a user defined function $f(e_1, \dots, e_m)$, defined by the ERLANG/EFM clauses

$$\begin{array}{l}
f(cp_{11}, \dots, cp_{1m}) \text{ when } g_1 \rightarrow ce_1; \\
\vdots \\
f(cp_{n1}, \dots, cp_{nm}) \text{ when } g_n \rightarrow ce_n.
\end{array}$$

is evaluated by first evaluating all arguments e_1, \dots, e_m to d_1, \dots, d_m . Similarly as with a case expression, each d_k is then matched against pattern cp_{jk} , and if all values d_1, \dots, d_m match patterns cp_{j1}, \dots, cp_{jm} the guard expression g_j is evaluated in the local environment ρ_t . For the first clause i with all patterns, cp_{i1}, \dots, cp_{im} matching and guard expression g_i evaluating to **true** in the local environment ρ' , the clause expression ce_i is evaluated.

$$\begin{array}{c}
\forall k : 1 \leq k \leq m : \langle e_k, \sigma_{k-1}, \rho_{k-1} \rangle \Rightarrow \langle d_k, \overline{b_k}, \sigma_k, \rho_k \rangle \\
\forall j, 1 \leq j < i : \text{match}(\overline{cp_j}, \overline{d}, \bullet, \rho_t) \text{ implies } \langle \sigma_m, \rho_t \rangle \not\vdash g_j \\
\text{match}(\overline{cp_i}, \overline{d}, \bullet, \rho') \\
\langle \sigma_m, \rho' \rangle \vdash g_i \\
\langle ce_i, \sigma_m, \rho' \rangle \Rightarrow \langle d, \overline{b}, \sigma', \rho'' \rangle \\
\hline
\langle f(e_1, \dots, e_m), \sigma_0, \rho_0 \rangle \Rightarrow \langle d, \overline{b_1} \dots \overline{b_m} \overline{b}, \sigma', \rho'' \rangle
\end{array}$$

Configuration access functions

A call to a configuration access function $f(e_1, \dots, e_m)$ where the name, f , is the name of a configuration access function, is evaluated by evaluating all arguments e_1, \dots, e_m to d_1, \dots, d_m , and thereafter applying the definition $\Delta(f)$ of f to the results. We include the configuration environment in the rule, since it is now relevant.

$$\begin{array}{c}
\langle e_1, \sigma_0, \rho_0 \rangle \Rightarrow \langle d_1, \overline{b_1}, \sigma_1, \rho_1 \rangle \\
\vdots \\
\langle e_m, \sigma_{m-1}, \rho_{m-1} \rangle \Rightarrow \langle d_m, \overline{b_m}, \sigma_m, \rho_m \rangle \\
\frac{f \in C \quad \Delta(f)(d_1, \dots, d_m) = d}{\Delta \models \langle f(e_1, \dots, e_m), \sigma_0, \rho_0 \rangle \Rightarrow \langle d, \overline{b_1} \dots \overline{b_m} \overline{b}, \sigma_m, \rho_m \rangle}
\end{array}$$

where f is a configuration access function name, whose meaning is defined in the configuration environment Δ .

Output expressions

An output expression $f(e_1, \dots, e_m)$ where f is an output event type, is evaluated by first evaluating all arguments e_1, \dots, e_m to d_1, \dots, d_m and then adding the resulting output event $f(d_1, \dots, d_m)$ to the sequence of generated output events.

$$\begin{array}{c}
\langle e_1, \sigma_0, \rho_0 \rangle \Rightarrow \langle d_1, \overline{b_1}, \sigma_1, \rho_1 \rangle \\
\vdots \\
\frac{f \in A_b \quad \langle e_m, \sigma_{m-1}, \rho_{m-1} \rangle \Rightarrow \langle d_m, \overline{b_m}, \sigma_m, \rho_m \rangle}{\langle f(e_1, \dots, e_m), \sigma_0, \rho_0 \rangle \Rightarrow \langle \text{true}, \overline{b_1} \dots \overline{b_m} f(d_1, \dots, d_m), \sigma_m, \rho_m \rangle}
\end{array}$$

2.5 Derived State Machines

Using the semantics of **ERLANG/EFM**, defined in Section 2.4, we can now describe how an **ERLANG/EFM** specification defines a state machine. Recall (from Section 1.3) that a state machine is defined by a set of locations and state variables, which together with a set of transitions define the reaction to input events.

An **ERLANG/EFM** specification defines, for each configuration environment, a state machine, whose components are defined as follows.

Locations

The *locations* are those declared by the `-locations` declaration; this also declares the initial location.

State variables

The *state variables* are those declared by the `-statevars` declaration.

State

A *state* is a tuple $\langle l, \sigma \rangle$ where l is a location, and σ is an environment with bindings of values to the state variables; the binding σ may be restricted by optional type declarations for the state variables. The *initial state* is the tuple $\langle l_0, \bullet \rangle$ where l_0 is the initial location and \bullet is the empty environment.

Transitions

The *transitions* are derived from the transition clauses by considering them as clauses that define a user defined function. Each unit of transition clauses of form

$$\begin{aligned} & l(a_0, u_1, \dots, u_m) \text{ when } g_1 \rightarrow ce_1; \\ & \quad \vdots \\ & l(a_0, u_1, \dots, u_m) \text{ when } g_n \rightarrow ce_n. \end{aligned}$$

defines a set of transitions as follows. Assume that for location l and input event $a(d_1, \dots, d_m)$, the transition rules for user defined functions, considering l as a user defined function, allow to derive the transition

$$\Delta \models \langle l(a, d_1, \dots, d_m), \sigma, \bullet \rangle \Rightarrow \langle \{\text{next_state}, l'\}, \bar{b}, \sigma', \rho \rangle,$$

i.e., according to the rule for user defined functions,

$l(a_0, u_1, \dots, u_m) \text{ when } g_i \rightarrow ce_i$ is the first transition clause, such that a_0 is a , and such that the guard expression g_i evaluates to `true` in the local environment where u_1, \dots, u_m are bound to d_1, \dots, d_m , and the corresponding clause expression ce_i evaluates to $\{\text{next_state}, l'\}$. Then there is a transition from the state $\langle l, \sigma \rangle$ to the state $\langle l', \sigma' \rangle$, triggered by the input event $a(d_1, \dots, d_m)$, and emitting the sequence \bar{b} of output events. We use the term *computation step* for a transition of the state machine under a certain configuration environment, and denote it as

$$\langle l, \sigma \rangle \xrightarrow{a(\bar{d})/\bar{b}} \langle l', \sigma' \rangle.$$

Each computation step $\langle l, \sigma \rangle \xrightarrow{a(\bar{d})/\bar{b}} \langle l', \sigma' \rangle$ denotes that the state machine receives an input event, performs some local computation and emits a (possibly empty) sequence of output events.

2.6 Runs, Traces, and Test Cases

Based on the definition of state machines and their computation steps, we can now define the input-output behavior of an **ERLANG/EFMS** specification (or, equivalently, its defined state machine) as follows.

A *run* of the state machine in a configuration environment Δ consists of a sequence of computation steps under Δ , of form

$$\langle l_0, \bullet \rangle \xrightarrow{a_1(\overline{d_1})/\overline{b_1}} \langle l_1, \sigma_1 \rangle \xrightarrow{a_2(\overline{d_2})/\overline{b_2}} \dots \xrightarrow{a_n(\overline{d_n})/\overline{b_n}} \langle l_n, \sigma_n \rangle$$

which starts in the initial state.

A *trace* is the finite sequence

$$a_1(\overline{d_1})/\overline{b_1} \quad a_2(\overline{d_2})/\overline{b_2} \quad \dots \quad a_n(\overline{d_n})/\overline{b_n}$$

of pairs, each consisting of an input event and sequence of output events, that label such a run. A *run over a trace* $a_1(\overline{d_1})/\overline{b_1} \dots a_n(\overline{d_n})/\overline{b_n}$ is a run whose labeling sequence is exactly this trace.

A *test case* is a pair

$$\langle a_1(\overline{d_1})/\overline{b_1} \quad \dots \quad a_n(\overline{d_n})/\overline{b_n}, \Delta \rangle$$

consisting of a trace $a_1(\overline{d_1})/\overline{b_1} \dots a_n(\overline{d_n})/\overline{b_n}$ and a configuration environment Δ . Intuitively, such a test case can be produced by choosing Δ as the configuration environment, and supplying the sequence $a_1(\overline{d_1}) \dots a_n(\overline{d_n})$ of input events. If the tested system conforms to the **ERLANG/EFMS** specification, then it should respond with the sequence $\overline{b_1} \dots \overline{b_n}$ of output events. This follows from the observation that the induced run of the **ERLANG/EFMS** specification is uniquely determined by the configuration environment Δ and the supplied sequence of input events, since the **ERLANG/EFMS** specification is deterministic.

2.7 Symbolic operational semantics

A central problem in test case generation is that a program has a very large number of possible inputs, which gives rise to a very large number of possible executions, and makes it difficult to select a manageable set of inputs for test cases. *Symbolic execution* [King 76] addresses this problem by representing inputs symbolically, which allows to represent the vast space of executions by a more modest set of symbolic executions; furthermore the set of symbolic executions can be used to guide the selection of inputs for test cases.

The key idea in symbolic execution is to use symbolic values instead of actual data values as input, and to let expressions and values contain symbolic values. In our work, symbolic values will mostly be used

to represent parameters of input events, and we will therefore refer to them as *symbolic parameters*. We must generalize the operational semantics to allow expressions containing symbolic parameters, thus defining a *symbolic semantics*. The symbolic semantics differs from the operational semantics of Section 2.4 in that expressions may contain symbolic parameters that are *not* evaluated to values. Since symbolic parameters are not evaluated, symbolic execution can be regarded as a partial evaluation [Consel 93] of ERLANG/EFM expressions. For example, an expression $1 + 2 + p_1$, which contains the symbolic parameter p_1 , is evaluated as much as possible to $3 + p_1$. The resulting expression, after symbolic execution, is an ERLANG/EFM *normal form expression*. Further, if the initial expression contains a user defined function, then this user defined function is evaluated, using its definition, as much as possible. Intuitively, ERLANG/EFM normal form expressions are ERLANG/EFM expressions that may contain symbolic parameters, but not if and case expressions, calls to user defined functions, and pattern matches.

The symbolic parameters of our symbolic semantics are either input expression parameters or configuration access functions. The values of these parameters are not known during test case generation, and are therefore treated symbolically. Different execution paths, resulting from different outcomes of (explicit or implicit) tests generate different symbolic executions. For each symbolic execution, a *path condition* is generated, which is the conditions, under which this particular execution can be performed; path conditions may depend on symbolic parameters. For example, a pattern match $\{0, u_1, 2\} = p_2$, where p_2 is a symbolic parameter, succeeds only if p_2 has a value such that

$$\text{is_tuple}(p_2) \text{ and } \text{size}(p_2) == 3,$$

and such that 0, u_1 and 2 match corresponding elements in p_2 . In order to represent such additional conditions, we extend the structural states of Section 2.4 by an extra component, which is the condition on symbolic parameters under which the structural state can be reached.

We define an ERLANG/EFM *normal form expression*, to be an expression which does not contain if and case expressions, calls to user defined functions, pattern matches, and bindings of state variables. We use ϵ to range over normal form expressions. Let \mathfrak{E} be the domain of ERLANG/EFM normal form expressions. A *symbolic parameter* is either

- an *input expression parameter*, i.e., a parameter p_i in an input expression $a(p_1, \dots, p_n)$ where a is an input event type, or
- a *configuration parameter*, i.e., an application $f(\epsilon_1, \dots, \epsilon_n)$ of a configuration access function f to ERLANG/EFM normal form expressions $\epsilon_1, \dots, \epsilon_n$.

Note that a configuration parameter may depend on input expression parameters. Thus, the value of a configuration parameter can be different for different combinations of arguments.

Definition 2.5 A *symbolic structural state* is a tuple

$$\langle e, g, \bar{\mathbf{b}}, \sigma, \rho \rangle$$

where

- e is an **ERLANG/EFMS** expression,
- g is an **ERLANG/EFMS** guard expression, which will not contain occurrences of local or global variables.
- $\bar{\mathbf{b}}$ is a sequence of output expressions, each of which is of form

$$b(\mathbf{e}_1, \dots, \mathbf{e}_n),$$

such that $b \in A_b$ is an output event type and $\mathbf{e}_1, \dots, \mathbf{e}_n$ are **ERLANG/EFMS** normal form expressions where each $\mathbf{e}_j \in \mathfrak{E}$,

- $\sigma \in V \rightarrow \mathfrak{E}$ is a global symbolic environment with bindings to the global state variables V , and
- $\rho \in U \rightarrow \mathfrak{E}$ is a local symbolic environment with bindings to the local variables U . \square

An empty (local or global) environment is denoted \bullet and the empty sequence is denoted ϵ . An *initial symbolic structural state* is the tuple

$$\langle a(\bar{p}), \text{true}, \epsilon, \bullet, \bullet \rangle$$

where $a(\bar{p})$ is an input expression, $a \in A_a$ is an input event type, and $\bar{p} \in U \cup D$ is a range of input expression parameters.

We always assume a transition to start in a symbolic structural state where we initially have an empty sequence of output expressions, and a guard expression **true**. For brevity, we will write such a symbolic structural state, of form $\langle e, \text{true}, \epsilon, \sigma, \rho \rangle$, as $\langle e, \sigma, \rho \rangle$.

A *symbolic transition* between two structural states

$$\langle e, \sigma, \rho \rangle \Rightarrow \langle \mathbf{e}, g, \bar{\mathbf{b}}, \sigma', \rho' \rangle$$

denotes that the expression e is evaluated to an **ERLANG/EFMS** normal form expression \mathbf{e} , in a global symbolic environment σ and local symbolic environment ρ , under the condition g . The evaluation generates the sequence $\bar{\mathbf{b}}$ of output expressions, σ' is the new global symbolic environment with the collected bindings of state variables, and ρ' is the new local symbolic environment with the collected bindings of local variables.

Note that, in contrast to the non-symbolic operational semantics defined in Section 2.4, a symbolic transition does not depend on any configuration environment. Instead, the resulting guard expression g may

include constraints on configuration access functions that occur in the symbolic transition. The symbolic transition then represents the set of (non-symbolic transitions) that are performed in an environment which satisfies the constraints in g .

The set of symbolic transitions is generated from *symbolic transition rules* of form:

$$\frac{cond_1, \dots, cond_m \quad trans_1, \dots, trans_n}{\langle e, \sigma, \rho \rangle \Rightarrow \langle \mathbf{e}, g, \bar{\mathbf{b}}, \sigma', \rho' \rangle}$$

where $cond_1, \dots, cond_m$ are conditions on when the symbolic transition rule can be applied, and $trans_1, \dots, trans_n$ are symbolic transitions on which the symbolic transition rule depends. A symbolic transition rule is *enabled* for an ERLANG/EFM expression e when all conditions are satisfied and all the symbolic transitions exist.

In the following subsections, we provide the symbolic semantics. Thereafter, in Section 2.8, we state and prove that the symbolic semantics is faithful to the operational semantics of Section 2.4.

2.7.1 Pattern matching

We define the relation $match(cp, \mathbf{e}, \rho, g', \rho')$ in Figure 2.4 to denote a successful pattern match between a pattern cp and an ERLANG/EFM normal form expression \mathbf{e} within a local symbolic environment ρ . In the case of a successful pattern match, g' holds additional conditions required for matching, and ρ' is the local symbolic environment updated with all new bindings created when matching. The match relation is undefined for all cp not matching \mathbf{e} . The definition is a straightforward modification of $match(cp, d, \rho, \rho')$ in Figure 2.3 where we additionally use some ERLANG functions. The ERLANG function `element($k, Tuple$)` returns the k :th element from tuple $Tuple$, `hd($List$)` returns the first element in $List$, `tail($List$)` returns a list with all elements except the first element in $List$, `is_list($List$)` returns `true` if $List$ is a list, `is_tuple($Tuple$)` returns `true` if $Tuple$ is a tuple, and `size($Tuple$)` returns the number of elements in $Tuple$.

2.7.2 Transition Rules for ERLANG/EFM Expressions

Basic and Compound values

All expressions in ERLANG/EFM can be evaluated to basic or compound values. In this section we give the symbolic transition rules for how ERLANG/EFM expressions are evaluated to ERLANG/EFM normal form expressions. The evaluation to values can then later be accomplished by

$$\begin{array}{c}
\frac{d \text{ is a basic or compound value}}{\text{match}(d, d, \rho, \text{true}, \rho)} \\
\\
\frac{u \text{ is an anonymous local variable}}{\text{match}(u, \epsilon, \rho, \text{true}, \rho)} \\
\\
\frac{u \text{ is a local variable, defined in } \rho}{\text{match}(u, \epsilon, \rho, \rho(u) == \epsilon, \rho)} \\
\\
\frac{u \text{ is a local variable, not defined in } \rho}{\text{match}(u, \epsilon, \rho, \text{true}, \rho[u \mapsto \epsilon])} \\
\\
\text{match}(cp_1, \text{element}(1, \epsilon), \rho_0, g_1, \rho_1) \\
\vdots \\
\text{match}(cp_n, \text{element}(n, \epsilon), \rho_{n-1}, g_n, \rho_n) \\
\hline
\text{match}\left(\{cp_1, \dots, cp_n\}, \epsilon, \rho_0, \left(\begin{array}{l} \text{is_tuple}(\epsilon) \\ \wedge \text{size}(\epsilon) == n \\ \wedge g_1 \wedge \dots \wedge g_n \end{array}\right), \rho_n\right) \\
\\
\frac{\text{match}(cp_1, \text{hd}(\epsilon_1), \rho_0, g_1, \rho_1) \quad \text{match}(cp_2, \text{tail}(\epsilon_2), \rho_1, g_n, \rho_2)}{\text{match}([cp_1|cp_2], \epsilon, \rho_0, \text{is_list}(\epsilon) \wedge g_1 \wedge g_2, \rho_2)} \\
\\
\frac{\text{match}(cp_1, \epsilon_1, \rho_0, g_1, \rho_1) \quad \dots \quad \text{match}(cp_n, \epsilon_n, \rho_{n-1}, g_n, \rho_n)}{\text{match}(\overline{cp}, \overline{\epsilon}, \rho_0, g_1 \wedge \dots \wedge g_n, \rho_n)}
\end{array}$$

Figure 2.4. Definition of the symbolic $\text{match}(cp, \epsilon, \rho, g', \rho')$ relation for a pattern match between a pattern cp and an ERLANG/EFMS normal form expression ϵ . Initially we have a guard expression true and a local environment ρ . After the pattern match we have a guard expression g' and a local environment ρ' . The guard expression g' holds the conditions required for matching, and the local environment ρ' holds ρ and all additional bindings of local variables that are generated by the matching.

assigning values to all symbolic parameters. As in the non-symbolic semantics given in Section 2.4.2, basic and compound values are expressions that evaluate to themselves. We also omit rules for records as these are represented by tuples. Non-empty tuples and conses are evaluated by first evaluating all sub-expressions in a left-to-right order.

$$\begin{array}{c}
\langle e_1, \sigma_0, \rho_0 \rangle \Rightarrow \langle \mathbf{e}_1, g_1, \overline{\mathbf{b}}_1, \sigma_1, \rho_1 \rangle \\
\vdots \\
\langle e_n, \sigma_{n-1}, \rho_{n-1} \rangle \Rightarrow \langle \mathbf{e}_n, g_n, \overline{\mathbf{b}}_n, \sigma_n, \rho_n \rangle \\
\hline
\langle \{e_1, \dots, e_n\}, \sigma_0, \rho_0 \rangle \Rightarrow \langle \{\mathbf{e}_1, \dots, \mathbf{e}_n\}, g_1 \wedge \dots \wedge g_n, \overline{\mathbf{b}}_0 \dots \overline{\mathbf{b}}_n, \sigma_n, \rho_n \rangle \\
\\
\frac{\langle e_1, \sigma_0, \rho_0 \rangle \Rightarrow \langle \mathbf{e}_1, g_1, \overline{\mathbf{b}}_1, \sigma_1, \rho_1 \rangle \quad \langle e_2, \sigma_1, \rho_1 \rangle \Rightarrow \langle \mathbf{e}_2, g_2, \overline{\mathbf{b}}_2, \sigma_2, \rho_2 \rangle}{\langle [e_1|e_2], \sigma_0, \rho_0 \rangle \Rightarrow \langle [\mathbf{e}_1|\mathbf{e}_2], g_1 \wedge g_2, \overline{\mathbf{b}}_1 \overline{\mathbf{b}}_2, \sigma_2, \rho_2 \rangle}
\end{array}$$

Local and State variables

Local variables are distinguished from state variables and must not previously be defined in ρ . Local variables are bound to **ERLANG/EFMS** normal form expressions by pattern matching.

$$\frac{\langle e, \sigma, \rho \rangle \Rightarrow \langle \mathbf{e}, g, \overline{\mathbf{b}}, \sigma', \rho' \rangle \quad \text{match}(cp, \mathbf{e}, \rho'', g', \rho')}{\langle cp = e, \sigma, \rho \rangle \Rightarrow \langle \mathbf{e}, g \wedge g', \overline{\mathbf{b}}, \sigma', \rho' \rangle}$$

A state variable is bound to a value by first evaluating an expression and then the global environment is updated with the result.

$$\frac{v \in V \quad \langle e, \sigma, \rho \rangle \Rightarrow \langle \mathbf{e}, g, \overline{\mathbf{b}}, \sigma', \rho' \rangle}{\langle v = e, \sigma, \rho \rangle \Rightarrow \langle \text{true}, g, \overline{\mathbf{b}}, \sigma'[v \mapsto \mathbf{e}], \rho' \rangle}$$

All variables must be bound to an **ERLANG/EFMS** normal form expression before they are used; the value of a state variable is found in the global symbolic environment, σ , and the value of a local variable is found in the local symbolic environment, ρ . A local variable has precedence over a global variable with the same name.

$$\frac{v \text{ is defined in } \sigma}{\langle v, \sigma, \rho \rangle \Rightarrow \langle \sigma(v), \text{true}, \epsilon, \sigma, \rho \rangle} \quad \frac{u \text{ is defined in } \rho}{\langle v, \sigma, \rho \rangle \Rightarrow \langle \rho(u), \text{true}, \epsilon, \sigma, \rho \rangle}$$

Built-in functions

All arithmetic expressions, type declarations and relative, logical operators and guard expressions are handled in the same way. For example,

the transition rule for the logical operator disjunction is

$$\frac{\begin{array}{c} \langle e_1, \sigma_0, \rho_0 \rangle \Rightarrow \langle \mathbf{e}_1, g_1, \overline{\mathbf{b}}_1, \sigma_1, \rho_1 \rangle \\ \vdots \\ \langle e_n, \sigma_{n-1}, \rho_{n-1} \rangle \Rightarrow \langle \mathbf{e}_n, g_n, \overline{\mathbf{b}}_n, \sigma_n, \rho_n \rangle \end{array}}{\langle \text{or } (e_1, \dots, e_n), \sigma_0, \rho_0 \rangle \Rightarrow \langle \text{or } (\mathbf{e}_1, \dots, \mathbf{e}_n), g, \overline{\mathbf{b}}_1 \dots \overline{\mathbf{b}}_n, \sigma_n, \rho_n \rangle}$$

where $g = g_1 \wedge \dots \wedge g_n$.

A sequence expression simply evaluates to the value of the last clause expression in the sequence.

$$\frac{\langle ce_0, \sigma, \rho \rangle \Rightarrow \langle \mathbf{e}_0, g_0, \overline{\mathbf{b}}_0, \sigma'', \rho'' \rangle \quad \langle ce_1, \sigma'', \rho'' \rangle \Rightarrow \langle \mathbf{e}_1, g_1, \overline{\mathbf{b}}_1, \sigma', \rho' \rangle}{\langle (ce_0, ce_1), \sigma, \rho \rangle \Rightarrow \langle (\mathbf{e}_0, \mathbf{e}_1), g_0 \wedge g_1, \overline{\mathbf{b}}_0 \overline{\mathbf{b}}_1, \sigma', \rho' \rangle}$$

If Expressions

An **if** expression is a sequential choice construct such that the evaluation proceeds with the first clause expression ce_i for which the corresponding guard expression g_i is evaluated to true. If several g_i are evaluated to true, the first one is chosen. Since each g'_i may contain symbolic parameters, the index i of the taken clause may not be calculated during symbolic execution. Thus, the conditions for evaluation order implies that negations of all clauses to $1, \dots, i-1$ must be added to the resulting guard expression. Further note that since guard expressions have a restricted syntax, their symbolic evaluation will not generate any guard expressions as the second component of the symbolic structural state: this simplifies the following rule.

$$\frac{\begin{array}{c} \langle g_1, \sigma, \rho \rangle \Rightarrow \langle g'_1, \text{true}, \epsilon, \sigma, \rho \rangle \\ \vdots \\ \langle g_i, \sigma, \rho \rangle \Rightarrow \langle g'_i, \text{true}, \epsilon, \sigma, \rho \rangle \\ \langle ce_i, \sigma, \rho \rangle \Rightarrow \langle \mathbf{e}, g'', \overline{\mathbf{b}}, \sigma', \rho' \rangle \end{array}}{\left\langle \left(\begin{array}{l} \text{if} \\ \quad g_1 \rightarrow ce_1; \\ \quad \vdots \\ \quad g_n \rightarrow ce_n \\ \text{end} \end{array} \right), \sigma, \rho \right\rangle \Rightarrow \langle \mathbf{e}, \neg g'_1 \wedge \dots \wedge \neg g'_{i-1} \wedge g_i \wedge g'', \overline{\mathbf{b}}, \sigma', \rho' \rangle}$$

Case Expressions

The **case** expression is somewhat complicated to handle, since the choice of executed clause depends on both that a pattern matches and that a guard evaluates to **true**. This choice may furthermore depend on symbolic

parameters, implying that we cannot uniquely determine the index i of the chosen clause. When evaluating a **case** expression, first an expression e is evaluated to an **ERLANG/EFMS** normal form expression \mathfrak{e}^e . In a left-to-right order, \mathfrak{e}^e is then matched against patterns cp_j , and if the match succeeds the guard expression g_j is evaluated. This combination of matches and guards may be unsuccessful for a number of clauses, until an i is found for which cp_i and \mathfrak{e}^e match and the guard expression g_i evaluates to **true**, allowing to evaluate the corresponding clause ce_i . Similary as with **if** expressions, since g^e , g'_i and g''_i may contain symbolic parameters, i may not be uniquely determined and additional conditions must be added to the resulting guard expression.

$$\begin{array}{l}
\langle e, \sigma, \rho \rangle \Rightarrow \langle \mathfrak{e}^e, g^e, \overline{b_1}, \sigma', \rho' \rangle \\
1 \leq i \leq n \\
\forall j \text{ such that } 1 \leq j < i : \\
\quad \text{if there are no } g'_j, \rho''_j \text{ s.t. } \text{match} \left(cp_j, \mathfrak{e}^e, \rho', g'_j, \rho''_j \right) \\
\quad \quad \text{then let } g'_j, g''_j, \text{ be false} \\
\quad \quad \text{else let } g'_j, \rho''_j, g''_j \text{ be s.t.} \\
\quad \quad \quad \text{match} \left(cp_j, \mathfrak{e}^e, \rho', g'_j, \rho''_j \right) \text{ and } \langle g_j, \sigma', \rho''_j \rangle \Rightarrow \langle g''_j, \text{true}, \epsilon, \sigma', \rho''_j \rangle \\
\quad \quad \text{match} \left(cp_i, \mathfrak{e}^e, \rho', g'_i, \rho''_i \right) \\
\quad \quad \langle g_i, \sigma', \rho''_i \rangle \Rightarrow \langle g''_i, \text{true}, \epsilon, \sigma', \rho''_i \rangle \\
\quad \quad \langle ce_i, \sigma', \rho''_i \rangle \Rightarrow \langle \mathfrak{e}, g''', \overline{b_2}, \sigma'', \rho''' \rangle \\
\quad \quad g = g^e \wedge \neg(g'_1 \wedge g''_1) \wedge \dots \wedge \neg(g'_{i-1} \wedge g''_{i-1}) \wedge (g'_i \wedge g''_i) \wedge g''' \\
\hline
\left\langle \begin{array}{l} \text{case } e \text{ of} \\ \quad cp_1 \text{ when } g_1 \text{ -> } ce_1 ; \\ \quad \vdots \\ \quad cp_n \text{ when } g_n \text{ -> } ce_n \\ \text{end} \end{array} \right\rangle, \sigma, \rho \rangle \Rightarrow \langle \mathfrak{e}, g, \overline{b_1 b_2}, \sigma'', \rho''' \rangle
\end{array}$$

To understand that this rule accurately reflects the non-symbolic rule, note that in the symbolic semantics there is, for each j , at most one combination of guard g'_j and environment ρ''_j for which $\text{match} \left(cp_j, \mathfrak{e}^e, \rho', g'_j, \rho''_j \right)$, and furthermore that the guard g''_j in the else branch of the rule is uniquely determined by g_j .

User Defined Functions

A call to a user defined function $f(e_1, \dots, e_m)$, defined by the ERLANG/EFSM clauses

$$\begin{aligned} & f(cp_{11}, \dots, cp_{1m}) \text{ when } g_1 \rightarrow ce_1; \\ & \vdots \\ & f(cp_{n1}, \dots, cp_{nm}) \text{ when } g_n \rightarrow ce_n. \end{aligned}$$

is evaluated in a similar way as a **case** expression. The main difference is that the matching in each clause is performed between two tuples of expressions. Otherwise, the selection of clause to evaluate is analogous to that for a **case** expression, and the rule is analogous.

$$\begin{aligned} & \forall k \text{ such that } 1 \leq k \leq m : \langle e_k, \sigma_{k-1}, \rho_{k-1} \rangle \Rightarrow \langle \mathbf{e}_k, g_k^e, \overline{\mathbf{b}}_k, \sigma_k, \rho_k \rangle \\ & 1 \leq i \leq n \\ & \forall j \text{ such that } 1 \leq j < i : \\ & \quad \text{if there are no } g'_j, \rho''_j \text{ s.t. } match(\overline{cp_j}, \langle \mathbf{e}_1, \dots, \mathbf{e}_m \rangle, \rho', g'_j, \rho''_j) \\ & \quad \text{then let } g'_j, g''_j, \text{ be false} \\ & \quad \text{else let } g'_j, \rho''_j, g''_j \text{ be s.t.} \\ & \quad \quad match(\overline{cp_j}, \overline{\mathbf{e}}, \rho', g'_j, \rho''_j) \text{ and } \langle g_j, \sigma', \rho'_j \rangle \Rightarrow \langle g''_j, \text{true}, \epsilon, \sigma', \rho''_j \rangle \\ & \quad match(\overline{cp_i}, \overline{\mathbf{e}}, \rho', g'_i, \rho''_i) \\ & \quad \langle g_i, \sigma', \rho'_i \rangle \Rightarrow \langle g''_i, \text{true}, \epsilon, \sigma', \rho''_i \rangle \\ & \quad \langle ce_i, \sigma', \rho'_i \rangle \Rightarrow \langle \mathbf{e}, g''', \overline{\mathbf{b}}, \sigma'', \rho''' \rangle \\ & \quad \frac{g = g_1^e \wedge \dots \wedge g_m^e \wedge \neg(g'_1 \wedge g''_1) \wedge \dots \wedge \neg(g'_{i-1} \wedge g''_{i-1}) \wedge (g'_i \wedge g''_i) \wedge g'''}{f(e_1, \dots, e_m), \sigma_0, \rho_0 \rangle \Rightarrow \langle \mathbf{e}, g, \overline{\mathbf{b}}_1 \dots \overline{\mathbf{b}}_m, \overline{\mathbf{b}}, \sigma', \rho''' \rangle} \end{aligned}$$

Configuration Access Functions

A call to a configuration access function $f(e_1, \dots, e_m)$ to access constant configuration data, is evaluated simply by evaluating all arguments e_1, \dots, e_m to $\mathbf{e}_1, \dots, \mathbf{e}_m$. This results in the application of a configuration access function to Erlang normal form expressions, which is a symbolic parameter that is not evaluated further.

$$\begin{aligned} & \langle e_1, \sigma_0, \rho_0 \rangle \Rightarrow \langle \mathbf{e}_1, g_1, \overline{\mathbf{b}}_1, \sigma_1, \rho_1 \rangle \\ & \vdots \\ & \frac{f \in C \quad \langle e_m, \sigma_{m-1}, \rho_{m-1} \rangle \Rightarrow \langle \mathbf{e}_m, g_m, \overline{\mathbf{b}}_m, \sigma_m, \rho_m \rangle}{\langle f(e_1, \dots, e_m), \sigma_0, \rho_0 \rangle \Rightarrow \langle f(\mathbf{e}_1, \dots, \mathbf{e}_m), g_1 \wedge \dots \wedge g_m, \overline{\mathbf{b}}_1 \dots \overline{\mathbf{b}}_m, \sigma_m, \rho_m \rangle} \end{aligned}$$

Output Expressions

A call to an output expression $f(e_1, \dots, e_m)$ where the name, f , is an output event type, is evaluated by first evaluating all arguments e_1, \dots, e_m

to $\mathfrak{e}_1, \dots, \mathfrak{e}_m$.

$$\begin{array}{c}
\langle e_1, \sigma_0, \rho_0 \rangle \Rightarrow \langle \mathfrak{e}_1, g_1, \overline{\mathfrak{b}}_1, \sigma_1, \rho_1 \rangle \\
\vdots \\
\langle e_m, \sigma_{m-1}, \rho_{m-1} \rangle \Rightarrow \langle \mathfrak{e}_m, g_m, \overline{\mathfrak{b}}_m, \sigma_m, \rho_m \rangle \\
f \in A_b \quad g = g_1 \wedge \dots \wedge g_m \\
\hline
\langle f(e_1, \dots, e_m), \sigma_0, \rho_0 \rangle \Rightarrow \langle \text{true}, g, \overline{\mathfrak{b}}_1 \dots \overline{\mathfrak{b}}_m, f(\mathfrak{e}_1, \dots, \mathfrak{e}_m), \sigma_m, \rho_m \rangle
\end{array}$$

2.8 Correspondence Between non-Symbolic and Symbolic Semantics

In this section, we make the relationship between the symbolic semantics and the original operational semantics more precise. For notational clarity, we will in this section use σ_s and ρ_s for symbolic (global and local) environments.

Recall that the (non-symbolic) operational semantics defines transitions of the form $\langle e, \sigma, \rho \rangle \Rightarrow \langle d, \bar{b}, \sigma', \rho' \rangle$, while the symbolic operational semantics defines transitions of the form $\langle e_s, \sigma_s, \rho_s \rangle \Rightarrow \langle \mathfrak{e}, g, \overline{\mathfrak{b}}, \sigma'_s, \rho'_s \rangle$, where the environments σ_s and ρ_s (and σ'_s and ρ'_s) bind variables to normal form expressions rather than values, and where \mathfrak{e} and $\overline{\mathfrak{b}}$ are normal form expressions. Normal form expressions may contain uninstantiated symbolic parameters (input expression parameters or applications of configuration access functions).

In order to relate normal form expressions to values, we must introduce mappings from symbolic parameters to values. We have already introduced configuration environments that map the set C of configuration access function names to functions. We must also introduce environments for input expression parameters. We therefore extend the concept of configuration environment to that of parameter environment. A *parameter environment* Γ is a mapping from input expression parameters to Erlang values and from C to functions. For a parameter environment Γ , let Δ_Γ denote its restriction to C . For a guard expression g which contains no local or global variables (but may contain symbolic parameters), we write $\Gamma \vdash g$ to denote that g evaluates to true in the environment Γ . This relationship can also be expressed using the operational semantics of Section 2.4, as $\Delta_\Gamma \models \langle \bullet, \bullet \rangle \vdash \Gamma(g)$.

We can now define a correspondence between (non-symbolic) and symbolic structural states. This will be embodied by a correspondence relation \triangleleft_Γ , which is parameterized on a parameter environment Γ .

Definition 2.6 For two ERLANG/EFM expressions e and e_s , where e_s may contain input expression parameters, but e does not, let $e \triangleleft_\Gamma e_s$ denote

that e can be obtained from e_s by a sequence of replacements of the following three types:

1. replacing an input expression parameter p by $\Gamma(p)$,
2. replacing an application $f(d_1, \dots, d_m)$ of a configuration access function f to Erlang values d_1, \dots, d_m by its result $\Gamma(f)(d_1, \dots, d_m)$ under Γ , and
3. replacing an application of a built-in function to Erlang values by its resulting value. \square

For a (non-symbolic) environment ρ and a symbolic environment ρ_s , let $\rho \triangleleft_\Gamma \rho_s$ denote that ρ and ρ_s have the same domain, and that $\rho(u) \triangleleft_\Gamma \rho_s(u)$ for all u in this domain. For a structural state $\langle e, \bar{b}, \sigma, \rho \rangle$, a symbolic structural state $\langle e_s, g, \bar{b}, \sigma_s, \rho_s \rangle$, and a parameter environment Γ , let

$$\langle e, \bar{b}, \sigma, \rho \rangle \triangleleft_\Gamma \langle e_s, g, \bar{b}, \sigma_s, \rho_s \rangle$$

denote that $\Gamma \vdash g$, that $e \triangleleft_\Gamma e_s$, and $\bar{b} \triangleleft_\Gamma \bar{b}$, and $\sigma \triangleleft_\Gamma \sigma_s$, and that $\rho \triangleleft_\Gamma \rho_s$. Recall that the condition g does not contain any global or local variables.

We should now establish that the symbolic operational semantics is faithful to the (non-symbolic) operational semantics. This correspondence is formulated in the following proposition.

Proposition 2.7 Assume that $\langle e, \epsilon, \sigma, \rho \rangle \triangleleft_\Gamma \langle e_s, \text{true}, \epsilon, \sigma_s, \rho_s \rangle$. Then

- (i) for each transition $\Delta_\Gamma \models \langle e, \sigma, \rho \rangle \Rightarrow \langle d, \bar{b}, \sigma', \rho' \rangle$ there is a symbolic transition $\langle e_s, \sigma_s, \rho_s \rangle \Rightarrow \langle \mathbf{e}, g, \bar{b}, \sigma'_s, \rho'_s \rangle$ such that $\langle d, \bar{b}, \sigma', \rho' \rangle \triangleleft_\Gamma \langle \mathbf{e}, g, \bar{b}, \sigma'_s, \rho'_s \rangle$,
- (ii) for each symbolic transition $\langle e_s, \sigma_s, \rho_s \rangle \Rightarrow \langle \mathbf{e}, g, \bar{b}, \sigma'_s, \rho'_s \rangle$ such that $\Gamma \vdash g$, there is a transition $\Delta_\Gamma \models \langle e, \sigma, \rho \rangle \Rightarrow \langle d, \bar{b}, \sigma', \rho' \rangle$ such that $\langle d, \bar{b}, \sigma', \rho' \rangle \triangleleft_\Gamma \langle \mathbf{e}, g, \bar{b}, \sigma'_s, \rho'_s \rangle$. \square

Intuitively, Proposition 2.7 says that the non-symbolic and the symbolic operational semantics have a rather tight correspondence. Property (i) states that each transition in the non-symbolic semantics has a corresponding transition in the symbolic semantics, and Property (ii) states that each transition in the symbolic semantics, which is consistent with the parameter environment in the sense that g is satisfied, has a corresponding transition in the non-symbolic semantics.

Proof of Proposition 2.7: Proposition 2.7 is established by structural induction, thereby establishing a corresponding relationship between concrete and symbolic operational semantics for each construct considered by the semantics. For most constructs, this is rather straight-forward.

For Pattern matching, the relationship must be slightly reformulated as follows. Assume that $cp \triangleleft_\Gamma cp_s$ and $\rho \triangleleft_\Gamma \rho_s$. Then

- (i) whenever $match(cp, d, \rho, \rho')$ then $match(cp_s, \mathbf{e}, \rho_s, g', \rho'_s)$ where $\Gamma \vdash g'$, and $d \triangleleft_\Gamma \mathbf{e}$, and $\rho' \triangleleft_\Gamma \rho'_s$,
- (ii) whenever $match(cp_s, \mathbf{e}, \rho_s, g', \rho'_s)$ with $\Gamma \vdash g'$, then $match(cp, d, \rho, \rho')$ where $d \triangleleft_\Gamma \mathbf{e}$, and $\rho' \triangleleft_\Gamma \rho'_s$.

Other constructs are treated in a similar way. Let us consider a complex case: that of **case** expressions. Consider a **case** expression of form

```

case  $e$  of
   $cp_1$  when  $g_1$   $\rightarrow$   $ce_1$ ;
   $\vdots$ 
   $cp_n$  when  $g_n$   $\rightarrow$   $ce_n$ 
end
```

and the transition rules in the non-symbolic operational semantics of Section 2.4 and in the symbolic semantics of Section 2.7.

We first consider Property (i) of the proposition. By induction, we can assume that the non-symbolic and symbolic evaluation of the expression e results in structural states $\langle d, \overline{b_1}, \sigma'', \rho'' \rangle$ and $\langle \mathbf{e}^e, g^e, \overline{b_1}, \sigma'_s, \rho'_s \rangle$ such that $\langle d, \overline{b_1}, \sigma'', \rho'' \rangle \triangleleft_\Gamma \langle \mathbf{e}^e, g^e, \overline{b_1}, \sigma'_s, \rho'_s \rangle$. By the inductive hypothesis, it follows that if $match(cp_j, d, \rho'', \rho_t)$, then there is g'_j and ρ''_{s_j} such that $match(cp_j, \mathbf{e}^e, \rho'_s, g'_j, \rho''_{s_j})$ and $\rho_t \triangleleft_\Gamma \rho''_{s_j}$. But now, from $\langle \sigma'', \rho_t \rangle \not\vdash g_j$, and $\sigma'' \triangleleft_\Gamma \sigma'_s$, and $\rho_t \triangleleft_\Gamma \rho''_{s_j}$, we infer by the inductive hypothesis, that $\langle g_j, \sigma'_s, \rho''_{s_j} \rangle \Rightarrow \langle g''_j, \text{true}, \epsilon, \sigma'_s, \rho''_{s_j} \rangle$ such that $\Gamma \not\vdash g''_j$, using the property that g_j has the simple form of a guard expression and can be evaluated in only one way. This establishes that there is a symbolic transition, which evaluates the same branch as the non-symbolic one. We can then again use the structural induction to infer that from $match(cp_i, d, \rho'', \rho''')$, $\langle \sigma'', \rho''' \rangle \vdash g_i$, and $\langle ce_i, \sigma'', \rho''' \rangle \Rightarrow \langle d, \overline{b_2}, \sigma', \rho' \rangle$, we can obtain $match(cp_i, \mathbf{e}^e, \rho'_s, g'_i, \rho''_{s_i})$, $\langle g_i, \sigma'_s, \rho''_{s_i} \rangle \Rightarrow \langle g''_i, \text{true}, \epsilon, \sigma'_s, \rho''_{s_i} \rangle$, and $\langle ce_i, \sigma'_s, \rho''_{s_i} \rangle \Rightarrow \langle \mathbf{e}, g''', \overline{b_2}, \sigma'', \rho''' \rangle$ such that

$$\langle d, \overline{b_2}, \sigma', \rho' \rangle \triangleleft_\Gamma \langle \mathbf{e}, g''', \overline{b_2}, \sigma'', \rho''' \rangle.$$

Property (ii) is established using a similar structural induction. □

2.9 Symbolic Runs and Test Cases

We can now define a symbolic version of the computation step-based semantics defined by an **ERLANG/EFM** specification, in analogy with the definitions in Section 2.5. Define a *symbolic state* to be a tuple $\langle l, \sigma, G \rangle$, where $l \in L$ is a location, σ is a symbolic environment with bindings to the state variables, and G is a boolean expression over symbolic parameters.

We will sometimes refer to G as a *path condition*. The *initial symbolic state* is the tuple $\langle l_0, \bullet, \text{true} \rangle$ where l_0 is the initial location in the EFSM and \bullet is the empty environment.

Definition 2.8 A *symbolic computation step* of the state machine is of form

$$\langle l, \sigma, G \rangle \xrightarrow{a(\bar{p})/\bar{b}} \langle l', \sigma', G' \rangle,$$

where

- $\langle l, \sigma, G \rangle$ is a symbolic state over a set of symbolic parameters disjoint from \bar{p} ,
- $a(\bar{p})$ is an input expression,
- \bar{b} is a sequence of output expressions,

such that, considering l as a user defined function,

$$\langle l(a, p_1, \dots, p_n), \sigma, \bullet \rangle \Rightarrow \langle \{\text{next_state}, l'\}, g, \bar{b}, \sigma', \rho \rangle$$

holds, where $\bar{p} = p_1, \dots, p_n$, and where G' is $G \wedge g$. □

Define a *symbolic run* of an EFSM to be a sequence of symbolic computation steps

$$\langle l_0, \bullet, \text{true} \rangle \xrightarrow{a_1(\bar{p}_1)/\bar{b}_1} \langle l_1, \sigma_1, G_1 \rangle \xrightarrow{a_2(\bar{p}_2)/\bar{b}_2} \dots \xrightarrow{a_n(\bar{p}_n)/\bar{b}_n} \langle l_n, \sigma_n, G_n \rangle$$

starting in the initial symbolic state. A *symbolic trace* of an EFSM is the finite sequence

$$a_1(\bar{p}_1)/\bar{b}_1 \quad a_2(\bar{p}_2)/\bar{b}_2 \quad \dots \quad a_n(\bar{p}_n)/\bar{b}_n$$

of pairs, each consisting of an input expression and a sequence of output expressions, that label such a symbolic run of the EFSM. Note that each output expression may depend on symbolic parameters, which are either preceding input expression parameters or applications of configuration access functions. We sometimes say that the symbolic run is a symbolic run *over* the symbolic trace induced by the run.

Definition 2.9 A *symbolic test case* is a tuple $\langle w, G \rangle$, where w is a symbolic trace and G is a path condition over the symbolic parameters of w . □

We can now extend the correspondence between non-symbolic and symbolic semantics from Section 2.8 to states and symbolic states.

For a parameter environment Γ , let $\langle l, \sigma \rangle \triangleleft_\Gamma \langle l, \sigma_s, G \rangle$ denote that $\Gamma \vdash G$ and $\sigma \triangleleft_\Gamma \sigma_s$. Then, using Proposition 2.7, we can derive a correspondence between nonsymbolic and symbolic computation steps of an EFSM, formulated in the following proposition.

Proposition 2.10 Let $\langle l, \sigma \rangle \triangleleft_{\Gamma} \langle l, \sigma_s, G \rangle$ for some parameter environment Γ . Then

- (i) whenever $\langle l, \sigma \rangle \xrightarrow{a(\bar{d})/\bar{b}} \langle l', \sigma' \rangle$ and $\bar{d} = \Gamma(\bar{p})$, then there is a symbolic computation step $\langle l, \sigma, G \rangle \xrightarrow{a(\bar{p})/\bar{b}} \langle l', \sigma', G' \rangle$ such that $\bar{b} \triangleleft_{\Gamma} \bar{\mathbf{b}}$ and $\langle l', \sigma' \rangle \triangleleft_{\Gamma} \langle l', \sigma'_s, G' \rangle$,
- (ii) whenever $\langle l, \sigma, G \rangle \xrightarrow{a(\bar{p})/\bar{b}} \langle l', \sigma', G' \rangle$ and $\Gamma \vdash G'$, then there is a computation step $\langle l, \sigma \rangle \xrightarrow{a(\bar{d})/\bar{b}} \langle l', \sigma' \rangle$ such that $\bar{d} = \Gamma(\bar{p})$ and $\bar{b} \triangleleft_{\Gamma} \bar{\mathbf{b}}$, and $\langle l', \sigma' \rangle \triangleleft_{\Gamma} \langle l', \sigma'_s, G' \rangle$. \square

Proposition 2.10 follows rather straight-forwardly from Proposition 2.7 and the definitions of non-symbolic and symbolic computation steps.

2.10 Defining and Normalizing Edge Clauses

An **ERLANG/EFMS** specification of a SUT may be structured in many ways. In particular, there are many ways to structure transition clauses. For instance, transition clauses for the same location and input event type can either be separate, using different guards, or joined by means of if or case expressions. Such differences make it difficult to define what it means for a test suite to cover the “edges” of a specification. To allow for a more robust definition of “edge”, we therefore define how to transform the transition clauses of a specification into a set of transition clauses on a restricted form, called *edge clauses*. This transformation uses the symbolic semantics. It amounts to deriving the edges in the state machine of Figure 1.2 (from the example in Section 1.3) from the **ERLANG/EFMS** specification of Figure 2.2 (from the example in Section 2.2).

Whenever the symbolic semantics generates a transition of form

$$\langle l(a, p_1, \dots, p_m), \sigma, \bullet \rangle \Rightarrow \langle \{\text{next_state}, l'\}, g, \bar{\mathbf{b}}, \sigma', \rho \rangle,$$

we can represent this as an *edge clause* of form

$$l(a_{k0}, u_{k1}, \dots, u_{km}) \text{ when } g'_k \rightarrow \overline{v_k = \mathbf{e}_k}, \overline{\mathbf{b}_k}, \{\text{next_state}, l'\}$$

where l is the location from which the edge originate, and for each clause with index k , for $k = 1, \dots, n$,

- a_{k0} is an input event type,
- u_{k1}, \dots, u_{km} are local variables that represent formal parameters to an input expression with arity m ,
- g'_k is a guard expression,
- $\overline{v_k = \mathbf{e}_k}$ are bindings to state variables,
- $\overline{\mathbf{b}_k}$ is a sequence of output expressions, and

- $\{\text{next_state}, l'\}$ is a tuple which represents a transition to the next location l' (which may be l).

It can be noted that this alternative representation of the state machine share similarities with e.g., Dijkstras guarded commands [Dijkstra 75] in the sense that each edge simply consists of a boolean guard and a sequence of simple commands, such as state variable bindings and output events.

In order to further uniformize the representation of transition clauses in an **ERLANG/EFSM** specification, we also define a transformation on edge clauses, which normalizes them by combining edge clauses with the same effect, but with potentially different guards.

Assume a set of **ERLANG/EFSM** edge clauses

$$\begin{aligned}
 & l(a_{10}, u_{11}, \dots, u_{1m}) \text{ when } g_1 \rightarrow \overline{v_1 = \mathfrak{e}_1}, \overline{\mathfrak{b}_1}, \{\text{next_state}, l'\} \\
 & \quad \vdots \\
 & l(a_{n0}, u_{n1}, \dots, u_{nm}) \text{ when } g_n \rightarrow \overline{v_n = \mathfrak{e}_n}, \overline{\mathfrak{b}_n}, \{\text{next_state}, l'\}
 \end{aligned}$$

representing edges from location l to location l' when triggered by an input event $a(d_1, \dots, d_m)$. Now, if all (or any subset) of these edge clauses have the same bindings to state variables (i.e., $\overline{v_1 = \mathfrak{e}_1}, \dots, \overline{v_n = \mathfrak{e}_n}$ are all equal to $\overline{v = \mathfrak{e}}$), and the same output expressions (i.e., $\overline{\mathfrak{b}_1}, \dots, \overline{\mathfrak{b}_n}$ are all equal to $\overline{\mathfrak{b}}$), they can be merged into a single edge clause, i.e.,

$$l(a, u_1, \dots, u_m) \text{ when } g_1 \vee \dots \vee g_n \rightarrow \overline{v = \mathfrak{e}}, \overline{\mathfrak{b}}, \{\text{next_state}, l'\}$$

If such a merge is possible, it limits the number of edge clauses and therefore also possibly the number of test cases selected for inclusion in a test suite. We will refer to a specification where all **ERLANG/EFSM** edge clauses are normalized as a *normalized specification* and the original specification, as it was written, as the *original specification*. See also the evaluation in Section 8.1 for some results and a discussion on normalizing a specification.

2.11 Creating an executable specification

The main purpose for creating an **ERLANG/EFSM** specification is for generating test suites. In addition, an **ERLANG/EFSM** specification rewritten into an executable **ERLANG** module, can be used for simulation of the specification or as a basis for an implementation of the specification. In the **ERLY MARSH** tool, we have implemented automated translation of an **ERLANG/EFSM** specification into an **ERLANG** module, see also Section 6.1. The resulting **ERLANG** module utilizes the popular `gen_fsm` behavior in **ERLANG/OTP** [Erl 15]. The translation of an **ERLANG/EFSM** specification into an **ERLANG** `gen_fsm` module is rather straightforward and summarized

in the rest of this section where we use the `ERLANG/EFsm` specification in Example 2.1 as a running example, the resulting `ERLANG` module can be found in Figure 2.5.

2.11.1 The `gen_fsm` behavior in `ERLANG/OTP`

To simplify implementing state machines in `ERLANG`, `ERLANG/OTP` has support for the `gen_fsm` behavior. This behavior allow to create `ERLANG` processes that realize a given state machine in a call-back module following certain rules. In particular the call-back module needs to export a function

- `init/1` called whenever the process is started,
- `terminate/1` called whenever the process is terminated, and
- one function for each location in the state machine, called whenever the state machine is in that location and a `gen_fsm` tagged `ERLANG` message is sent to the `ERLANG` process.

In `ERLANG`, processes are started and stopped dynamically. To use the `gen_fsm` behavior, an `ERLANG` process can be started with a call to `gen_fsm:start_link/3`, which in turn will call `init/1` to initialize the process. After initialization, `gen_fsm` tagged messages can be sent to the `ERLANG` process with calls to `gen_fsm:send_event/2` (for asynchronous coomunication), which in turn will call a function sharing the name of the current location. A state transition to a location `otherloc` is then possible whenever the incoming `ERLANG` message matches the arguments of the call-back function in the current location and this function returns a tuple of form `{next_state,otherloc,State}` where `State` contains the values of all state variables passed to the next state, typically represented as a record.

A running `ERLANG` process is terminated by letting the call-back function used for state transitions return a tuple `{stop,Reason,State}`, where `Reason` is the reason for termination. The `gen_fsm` behavior further implements support for two different communication models; synchronous in which the calling party is waiting for a response before continuing execution, and asynchronous in which the calling party is not waiting for a response before continuing execution.

2.11.2 Creating an executable `ERLANG` module

To create an executable `ERLANG` module, utilizing the `gen_fsm` behavior we start with a number of mandatory declarations.

```
%% -- Test module begin --
-module(test_erlsrc).
-behaviour(gen_fsm).
```

```
-export([start_link/0,init/1,terminate/1]).
```

Here we give the `ERLANG` module the name `test_erlsrc` and export functions to start and terminate an `ERLANG` process. We also need to export a function for each location in the specification. In Section 2.3.2, three types of locations were distinguished in the `locations` declaration; *startloc* as the start location, *stoplocs* as the stop locations and *otherlocs* as all other locations. From Example 2.1 we have

```
-locations({morning,[end_of_day],
             [workUU, workMA, preschool]}).
```

and, as we are interested in asynchronous communication here, we create declarations of the locations as functions with 2 arguments.

```
-export([morning/2, workUU/2, workMA/2, preschool/2]).
```

Note that the stop location `end_of_day` is not exported. This is due to that at the stop location the execution of a run of the EFSM is supposed to stop and a way to achieve this is to simply restart the process. We do this by replace any transition to the stop location with a request to stop the process (by letting the call-back function return e.g., `{stop,normal,State}`). Then, we assume the process to be *supervised* by a supervisor (feature of `ERLANG/OTP`) that immediately restarts the process.

State variables

With the special treatment of state variables as global variables in `ERLANG/EFsm` all (guard) expressions with a state variable need to be rewritten. We let state variables be stored in a record in the resulting `ERLANG` module. The record declarations in `ERLANG/EFsm` use `ERLANG` syntax and can be declared as is. But we also define an additional record to hold all state variables. The name of this record must not conflict with any other record definition in the `ERLANG/EFsm` specification. Here we will use a record `state` to hold the state variables.

```
-record(state, {'Progress','Stamina','Day'}).
```

where we define each field in the record with the same name as a corresponding state variable. Thus, usage of a state variable is rewritten to access of a field in the `state` record. For example, usage of a state variable `Progress` can be rewritten to `Tmp0#state.'Progress'`.

In `ERLANG/EFsm` the old value of a state variable is destructively destroyed when assigned a new value. This is not possible in `ERLANG` and we must therefore use temporary variables, with unique names, to keep intermediate values. For example,

```
Progress=Progress+1
```

is rewritten to

```
Tmp1=Tmp0#state{'Progress'=Tmp0#state.'Progress'+1}
```

where Tmp1 is a new temporary variable holding the updated **state** record. When translating an **ERLANG/EFSM** transition clause, the last update of a state variable is translated to a temporary variable holding the **state** record returned by the call-back function.

In **ERLANG/EFSM** state variables can be used and bound in user defined functions. This is handled by adding an additional argument to all user defined functions, holding the **state** record, and let the return value always include an updated **state** record.

Input events

The **gen_fsm** call-back functions are created from **ERLANG/EFSM** transition clauses. The first argument, representing an incoming event, is created by creating a tuple of the input event type and all parameters associated with the input event. The second argument represents the current values of the state variables. In this way all transition clauses, for all input events that can occur at a location, are represented by a single call-back function. If multiple sets of transition clauses are defined at a location with the same arity, they are simply concatenated with a semicolon. All expressions of each transition clause must then be examined for any occurrence of a configuration access function, output event, or binding or usage of a state variable. If such an expression is found a rewrite is necessary as further explained in Section 2.11.2 and Section 2.11.2. For example,

```
morning(wakeup,TDay) ->
    Progress=0,
    Stamina=2,
    Day=TDay,
    if
        daytype(Day)==collect ->
            checkout(Day),
            {next_state,workUU};
        daytype(Day)==leave ->
            {next_state,preschool}
    end.
```

is rewritten to

```
morning({wakeup,TDay},Tmp0) ->
    Tmp1=Tmp0#state{'Day'=TDay,'Stamina'=2,'Progress'=0},
    Tmp2=Tmp1#state.'Day',
    case exampleIF_proto:daytype(Tmp2) of
        Tempdaytype0 when Tempdaytype0 == collect ->
```

```

        exampleIF_proto:checkout(Tmp2),
        {next_state, workUU, Tmp1};
    Tempdaytype0 when Tempdaytype0 == leave ->
        {next_state, preschool, Tmp1}
end.

```

where Tmp0 is an ERLANG variable holding the state record, checkout(Day) is an output event, Progress, Stamina and Day are state variables, and daytype(Day) is a configuration access function.

Output events and configuration access functions

Configuration access functions and output events are not further defined by the ERLANG/EFSM specification. Instead we assume some other ERLANG module (e.g., testIF) to implement these functions. To access the implementation, each output event outevent(Arg1) is rewritten to testIF:outevent(Arg1).

Predicates that can be used in guards in ERLANG are restricted to a well-defined set that cannot be extended. Thus, whenever we want to use a configuration access function in a guard it must be rewritten as a case expression such that the value can be assigned to a unique temporary variable and the guard rewritten as a legal ERLANG guard expression. For example, if config(Arg1) is a configuration access function then

```

loc(inevent,Arg1) when config(Arg1)==some_thing ->
    {next_loc,done};
loc(inevent,Arg1) when config(Arg1)==other_thing ->
    {next_loc,otherloc}.

```

is rewritten to

```

loc({inevent, Arg1}, State) ->
    case testIF:config(Arg1) of
        Tempconf0 when Tempconf0 == some_thing ->
            {stop, normal, State};
        Tempconf0 when Tempconf0 == other_thing ->
            {next_state, otherloc, State}
    end.

```

where loc is a stop location and Tempconf0 a new temporary variable.

ERLANG process management

After all transition clauses have been processed we add functions for handling starting and stopping an ERLANG process. To start a new process we define start_link/0 as

```

start_link() -> gen_fsm:start_link(test_erlsrc, [], []).

```


where the first argument specifies the name of the call-back module, the second argument is the argument forwarded to the call-back used for initialization of the created process, and the third argument is a list of options (i.e., no options here). The `init` function is called when the `ERLANG` process is initialized and is here declared as

```
init([]) -> {ok, morning, #state{}}.
```

where the initial location is set to `morning` and the `state` record is The initial state data of the `gen_fsm`. Finally, we need to add the call-back function `terminate/3`.

```
terminate(_Reason, _StateName, _StateData) -> ok.
```

The call-back function is mandatory and used when an `ERLANG` process is terminated, but here it do nothing

For sending an input event, e.g., `wakeup(TDay)`, to a running `ERLANG` process implemented as in Example 2.1 we do

```
gen_fsm:send_event(Pid, {wakeup, TDay}).
```

where `Pid` is the process identifier as returned by `start_link/0`.

In the example, the declarations of the user defined types found in the original `ERLANG/EFsm` specification have been excluded. This is typically what is desired if the purpose of the rewriting is to form the basis for an implementation as, in general, the type declarations used on the abstract level in the specification does not apply. If the purpose of the translation is to execute the specification, these type declarations can be kept, as is, from the `ERLANG/EFsm` specification.

```

-module(test_erlsrc).
-behaviour(gen_fsm).
-export([start_link/0,init/1,terminate/3]).
-export([morning/2, workUU/2, workMA/2, preschool/2]).
-record(state, {'Progress','Stamina','Day'}).

morning({wakeup, TDay}, Tmp0) ->
    Tmp1=Tmp0#state{'Day'=TDay,'Stamina'=2,'Progress'=0},
    Tmp2=Tmp1#state.'Day',
    case exampleIF_proto:daytype(Tmp2) of
        Tempdaytype0 when Tempdaytype0 == collect ->
            exampleIF_proto:checkout(Tmp2),
            {next_state, workUU, Tmp1};
        Tempdaytype0 when Tempdaytype0 == leave ->
            {next_state, preschool, Tmp1}
    end.

workUU({progress, X}, Tmp0) when is_integer(X) ->
    Tmp1 = Tmp0#state{'Progress'=Tmp0#state.'Progress' + X},
    if Tmp1#state.'Progress' >= 8 ->
        exampleIF_proto:checkin(Tmp1#state.'Day'),
        {stop,normal,Tmp1};
    true -> {next_state, workUU, Tmp1}
end;

workUU({incident, I}, Tmp0) ->
    Tmp1=Tmp0#state{'Stamina'=Tmp0#state.'Stamina' - 1},
    exampleIF_proto:checkin(Tmp1#state.'Day'),
    if (Tmp1#state.'Stamina'=<0) or (I==pernilla_call) -> {stop,normal,Tmp1};
    I==bug -> {next_state, workMA, Tmp1};
    I==kids_notok -> testIF:checkin(Tmp1#state.'Day'),
        {next_state, preschool, Tmp1}
end.

workMA({progress, X}, Tmp0) when is_integer(X) ->
    Tmp1 = Tmp0#state{'Progress'=Tmp0#state.'Progress' + X},
    if Tmp1#state.'Progress' >= 8 -> {stop,normal,Tmp1};
    true -> {next_state, workMA, Tmp1}
end.

preschool({incident, I}, Tmp0) ->
    Tmp1 = Tmp0#state{'Stamina'=Tmp0#state.'Stamina' - 1},
    if (Tmp1#state.'Stamina'=<0) or (I==kids_notok) -> {stop,normal,Tmp1};
    I==kids_happy -> exampleIF_proto:checkout(Tmp1#state.'Day'),
        {next_state, workUU, Tmp0}
end.

init([]) -> {ok, morning, #state{}}.
start_link() -> gen_fsm:start_link(test_erlsrc, [], []).
terminate(_Reason, _StateName, _StateData) -> ok.

```

Figure 2.5. An ERLANG/OTP gen_fsm implementation of the ERLANG/EFsm specification from Example 2.1.

3. Specifying test case selection

In this chapter, we present a technique for specifying test case selection in a simple and flexible manner.

In the previous chapter, we introduced `ERLANG/EFSM`, a specification language for modeling systems under test as extended finite state machines (EFSMs). Test cases can be extracted from runs of such EFSMs. Since models of realistic systems typically give rise to a very large number of possible test cases, this poses the problem of how to select test cases to form test suites. A common way to address this problem is to define *coverage criteria*. These can be seen as desirable properties of test suites that are generated from the formal specification, which force the test suite to exercise different aspects of the specification. For specifications of SUTs as state machines, classical examples of coverage criteria include coverage of all locations, coverage of all transitions, and coverage of all definition-use pairs. Various other criteria for test suites, such as test purposes [Fernandez 97, Rusu 00], or criteria that focus on individual state variables or components of control flow [Friedman 02], can also be seen as coverage criteria.

Different coverage criteria are suitable in different situations, and satisfy different constraints on fault detection capability, cost, information about where potential faults may be located, etc. Thus, it is highly desirable that a test generation tool is able to generate test suites in a flexible manner, for a wide variety of different coverage criteria. In other words, a test generation tool should accept a simple specification of a coverage criterion, given in a language that can easily specify a large set of coverage criteria, and be able to generate test suites accordingly.

In our technique, a coverage criterion is given as a set of *coverage items*. Each coverage item represents a certain property of a test case. Recall that a test case can be seen as a sequence of input and output events generated by traversing the EFSM in a certain way. A coverage item is thus a property of such a traversal. For instance, a coverage item can state that a particular state, edge, or combination thereof, should be visited during the traversal, it can also be an explicit test purpose, etc. Using techniques from model checking and run-time monitoring [Vardi 86, Havelund 02], we can specify a coverage item by an automaton, which we call an *observer*, which observes the execution of a test case, and reports acceptance when the test case has *covered* the coverage item that it specifies. For instance, a coverage item stating that a control state

l of an EFSM model should be visited simply observes how the EFSM executes and reports acceptance when it enters l .

A typical coverage criterion is given as a (often rather large) set of coverage items. An important mechanism to facilitate specification of many coverage criteria is to allow *parameterization* of observers. In this way, one can specify a set of coverage items parameterized over, e.g., control states, data variables, edges, etc. of the EFSM model. Using this simple and general mechanism, we can specify most of the coverage criteria that have been used in the literature, and also tailor coverage to specific features of a particular SUT. For instance, if a particular interface is very error prone, we can specify a coverage criterion which requires all possible interleavings of interactions on that interface to be exhibited in a test suite.

In Chapter 4, we give examples on how several commonly used coverage criteria can be defined using observers, and in Chapter 5 we present techniques for generating test suites from **ERLANG/EFMS** models extended with observers.

This chapter is organized as follows. Section 3.1 gives an informal introduction to observers. Section 3.2 introduces **ERLANG/OBS** as a language for specifying observers, and describes the main syntactic constructs. Section 3.3 describes an important mechanism of **ERLANG/OBS**, which allow to introduce user defined observer predicates. Section 3.4 introduces an alternative graphical notation for specifying observers. Section 3.5 describes the operational semantics for observers in **ERLANG/OBS**, which is used in Section 3.6 to explain how an observer is derived from an **ERLANG/OBS** description. Section 3.7 describes a symbolic semantics, intended for symbolic test case generation, and its correspondence to the semantics of Section 3.5. Section 3.8 provides a symbolic version of the definition of observer from Section 3.6, and defines a symbolic notion of coverage; the chapter concludes with Theorem 3.17, which establishes a correspondence between test cases, coverage, and symbolic runs, that will be exploited by the test case generation algorithms in Chapter 5

3.1 Observers: An Informal Introduction

In this section, we informally introduce observer automata (observers) as a tool to specify coverage criteria defined in terms of coverage items, to be used in test suite generation. Letting a test case correspond to an execution of the EFSM, we can use techniques from model checking and run-time verification [Vardi 86, Havelund 02] to represent a coverage item by an *observer*. An observer monitors how an EFSM executes a run over a trace, and keeps track of some chosen aspects of the EFSM execution. The observer can observe the events and values of global state variables

of the run, as well as syntactical components of edges that the EFSM traverses in response to observed events, but must not interfere with the execution of the system. It “accepts” whenever a represented coverage item has been covered. In contrast to the EFSMs defined by `ERLANG/EFSM`, observers may be non-deterministic, since a particular coverage item may be covered in several ways.

As a very simple example, the coverage item “visit location l of the EFSM” can be represented by an observer with one initial state, which monitors the EFSM until the location l is entered, and one accepting state, which is entered when the EFSM enters location l . This can be specified

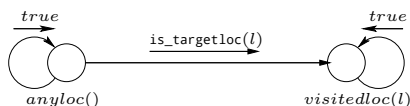


Figure 3.1. An observer representing the coverage item “visit location l of the EFSM” in the $visitedloc(l)$ state.

using a graphical notation as in Figure 3.1 where $anyloc()$ represents the initial observer state and $visitedloc(l)$ the accepting observer state. The observer makes a transition from $anyloc()$ to $visitedloc(l)$ whenever the observer guard $is_targetloc(l)$ is true, which occurs exactly when the target location of a computation step in the EFSM is l . We let each accepting observer state represent a coverage item. Thus, when the accepting observer state is entered, the execution has covered the corresponding coverage item. In the following, we will often describe simple coverage items, such as this one, without explicitly mentioning the verb “visit”, e.g., we say coverage item “location l ”, meaning “visit location l ”.

Typical coverage criteria consist not only of a single coverage item, but of a large set of coverage items which are often similar, but distinguished by some parameters. A parameter can, e.g., be a target location, an edge, or a state variable. We therefore use observers with many accepting states so that we can specify a *set* of coverage items. The coverage items “all

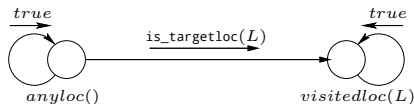


Figure 3.2. An observer representing the coverage item “all target locations of the EFSM” in the $visitedloc(L)$ state.

target locations of the EFSM” can be represented by a nondeterministic observer with one initial state, and many accepting states, one for each possible target location in the EFSM. The initial state monitors the EFSM until some target location occurs in the EFSM, and an accepting state

can be entered after any target location of the EFSM. Now, each edge in the EFSM has a single target location so in this case a single accepting state will be entered after each edge in the EFSM.

This can be specified using a graphical notation as in Figure 3.2 where *anyloc()* represents the initial observer state and *visitedloc(L)* the many accepting observer states. The observer guard *is_targetloc(L)* evaluates to true whenever the EFSM enter the target location of a computation step in the EFSM binds a state variable and additionally binds L to the target location.

Thus, for each target location the observer can make a transition from *anyloc()* to *visitedloc(L)*. To allow an observer to start monitoring at any point of an EFSM run, a self-loop is assumed in the initial observer state. Similarly, to remember already covered items, we let any accepting observer state have a self-loop with an observer guard *true* associated. In Section 3.8, we discuss the effect of these self-loops in more detail.

Let us now formalize the concept of observer as a nondeterministic automaton which accepts runs of an EFSM. Recall from Section 2.5 that a run of an EFSM is a sequence of computation steps

$$\langle l_0, \bullet \rangle \xrightarrow{a_1(\overline{d_1})/\overline{b_1}} \langle l_1, \sigma_1 \rangle \xrightarrow{a_2(\overline{d_2})/\overline{b_2}} \dots \xrightarrow{a_n(\overline{d_n})/\overline{b_n}} \langle l_n, \sigma_n \rangle,$$

each of which is derived from a unique transition clause of the specification. Since the observer often depends on syntactic elements of such a transition clause (e.g., state variables that are bound to values), we will annotate the computation step by this transition clause. In fact, our annotation will consist only of that branch of the transition clause which is traversed to generate the computation step. Recall, from Section 2.10, that such a branch can be represented as an **ERLANG/EFMS** *edge clause*. Thus, we define an *annotated computation step* as a pair

$$\left\langle \begin{array}{l} \langle l, \sigma \rangle \xrightarrow{a(\overline{d})/\overline{b}} \langle l', \sigma' \rangle, \\ l(a, u_1, \dots, u_m) \text{ when } g \rightarrow \overline{v} = \overline{c}, \overline{b}, \{\text{next_state}, l'\} \end{array} \right\rangle$$

consisting of

- a computation step $\langle l, \sigma \rangle \xrightarrow{a(\overline{d})/\overline{b}} \langle l', \sigma' \rangle$, and
- the unique **ERLANG/EFMS** edge clause, of form

$$l(a, u_1, \dots, u_m) \text{ when } g \rightarrow \overline{v} = \overline{c}, \overline{b}, \{\text{next_state}, l'\} ,$$

from which the computation step is derived.

Observers can now be formalized as automata over annotated computation steps. We define an *observer* as a tuple $\langle \Sigma, Q, q_0, Q_f, \rightarrow \rangle$ where

- Σ , the input alphabet for an observer, is the set of annotated computation steps of the **ERLANG/EFMS** specification.

- Q , ranged over by q , is a finite set of *observer states*. An observer state is of form $\iota(d_1, \dots, d_n)$, where $\iota \in L^o$ is an observer location, and $d_1, \dots, d_n \in D$ are values.
- q_0 is the *initial observer state*.
- $Q_f \subseteq Q$ is the set of *accepting observer states*, representing coverage items.
- $\rightarrow \subseteq Q \times \Sigma \times Q$ is the *observer step relation*. We use $q \xrightarrow{\varsigma} q'$ to denote $\langle q, \varsigma, q' \rangle \in \rightarrow$ where ς is an annotated computation step, and q and q' are observer states.

Intuitively, one can think of an observer as an automaton which accepts runs in the usual way; whenever it is in one of its observer states, q say, and the EFSM performs a computation step at the occurrence of an input event, the corresponding annotated computation step ς is generated as input to the observer, which makes an observer step of form $q \xrightarrow{\varsigma} q'$ thereby moving to the state q' .

We can then define what it means for a run or test case to cover a certain coverage item. Let $\langle l_0, \bullet \rangle \xrightarrow{a_1(\overline{d_1})/\overline{b_1}} \dots \xrightarrow{a_n(\overline{d_n})/\overline{b_n}} \langle l_n, \sigma_n \rangle$ be a run of an EFSM, and let for $i = 1, \dots, n$, ς_i represent the i th annotated computation step

$$\left\langle \begin{array}{l} \langle l_{i-1}, \sigma_{i-1} \rangle \xrightarrow{a_i(\overline{d_i})/\overline{b_i}} \langle l_i, \sigma_i \rangle, \\ l_{i-1}(a, u_1, \dots, u_m) \text{ when } g \rightarrow \overline{v} = \overline{e}, \overline{b}, \{\text{next_state}, l_i\} \end{array} \right\rangle$$

Then we say that the run *covers* a coverage item $q_f \in Q_f$ if there is a run

$$q_0 \xrightarrow{\varsigma_1} q_1 \xrightarrow{\varsigma_2} q_2 \cdots q_{n-1} \xrightarrow{\varsigma_n} q_n$$

of the observer over $\varsigma_1 \cdots \varsigma_n$ which ends in $q_f = q_n$. We say that a test case $\langle a_1(\overline{d_1})/\overline{b_1} \cdots a_n(\overline{d_n})/\overline{b_n}, \Delta \rangle$ *covers* the coverage item q_f if the run induced by the test case covers q_f .

3.2 ERLANG/OBS - observers with ERLANG syntax

In this and the following section we introduce ERLANG/OBS as a language to describe observers. Similarly to ERLANG/EFSM, presented in Section 2.3, ERLANG/OBS uses a restricted part of the ERLANG syntax, with some additional support for declaration of observers.

Our ERLANG/OBS language supports two classes of variables: *observer variables* and *match variables*. Observer variables function similarly as local variables in ERLANG/EFSM. Match variables are a particular class of variables, whose purpose is to make the current annotated computation step of an EFSM accessible to an observer defined in ERLANG/OBS. Match

variables assume values that are different components of the current annotated computation step. For instance, the match variable `TargetLoc` is always assigned to the target location of the current annotated computation step. This allows to assign the target location also to an observer variable, e.g., by pattern matching with the match variable `TargetLoc`. For convenience, `ERLANG/OBS` includes a number of built-in *observer predicates* that perform common such tasks. For instance, the observer predicate `is_targetloc(L)` matches `L` with the current value of the match variable `TargetLoc`. In `ERLANG/OBS`, such observer predicates are used in guards of observer clause expressions, which define observer step, in the same way as transition clauses define computation steps in `ERLANG/EFsm`. Observer edge clauses in `ERLANG/OBS` have many similarities with edge clauses in `ERLANG/EFsm`, but have a simpler form: they cannot contain `if` and `case` expressions or calls to user defined functions.

<code>-obs_locations({ι_0, stop, other}).</code>	Declarations of observer locations
<code>-obs_usage(ι_0, ous).</code>	Declarations of observer usage
<code>-obs_predicates([$f_1^{op}/i_1, \dots, f_n^{op}/i_n$]).</code>	Declarations of user defined observer predicates
<code>$\iota(u_1, \dots, u_m)$ when $h_1 \rightarrow oce_1$;</code> \vdots	Observer edge clauses for all declared observer locations
<code>$\iota(u_1, \dots, u_m)$ when $h_n \rightarrow oce_n$.</code>	

Figure 3.3. An overview of a typical `ERLANG/OBS` specification.

The overall structure of an `ERLANG/OBS` specification is given in Figure 3.3. It consists of

- a declaration of observer locations,
- a declaration of usage,
- a declaration of user defined observer predicates, and
- for each observer location, a list of *observer edge clauses* representing all observer edges in the observer.

To illustrate, we begin with two examples of `ERLANG/OBS` observers.

Example 3.1 An observer representing the coverage items “all target locations of the *EFsm*” can be specified in `ERLANG/OBS` as

```
-obs_locations({anyloc,[visitedloc],[ ]}).
```



```

anyloc() when is_targetloc(L) ->
    {next_state,visitedloc(L)}.

```

This specification says that we have an initial observer location, `anyloc`, and an accepting observer location, `visitedloc`. The observer guard `is_targetloc(L)` is an observer predicate that matches the observer variable `L` with the current value of the match variable `TargetLoc`. Thus, transition from `anyloc` to `visitedloc` occurs when the EFSM executes an annotated computation step, and `L` matches the value of `TargetLoc`. As `L` here is a previously unbound observer variable, the match will always succeed, `L` will be bound to the target location in the EFSM, and `is_targetloc(L)` will return `true`. Further, for all target locations `l` of the EFSM a coverage item will be represented by `visitedloc(l)`. From `anyloc` and `visitedloc` there exist implicit transitions to the location itself. Due to these self-loops, each observed target location can generate a new coverage item and previously generated coverage items are recorded. □

Example 3.2 A common coverage criterion is coverage of all definition-use pairs. A run of an `ERLANG/EFM` specification covers a definition-use pair if an assignment (definition) of a state variable is followed by a usage of the same state variable. We represent a definition-use pair as a triple that consists of a state variable, an edge in the EFSM where the state variable is defined, and another edge in the EFSM where the state variable is used. For example, in Example 1.1, there is a definition-use pair for the state variable *Progress* defined on the edge between the `morning` and `workUU` locations, and used on one of the edges between the `workUU` and `end_of_day` locations. An observer for *any* definition-use triplet with a state variable and two edges, can be specified in `ERLANG/OBS` as

```

-obs_locations({q,[du],[q1]}).

```

```

q() when is_definedvar(V),is_edge(E) ->
    {next_state,q1(V,E)}.

```

```

q1(V,E) when not is_definedvar(V) ->
    {next_state,q1(V,E)};
q1(V,E) when is_usedvar(V),is_edge(F) ->
    {next_state,du(V,E,F)}.

```

Here `q` is the initial observer location, `q1` an observer location entered whenever a binding of some state variable `v` on some edge in the EFSM has occurred, and `du` an accepting observer location entered whenever the same state variable is also used on some other edge. The clauses contain a number of observer predicates that evaluate tests over match variables and provided observer variables. In `q` we test for definition of

some state variable, and the existence of an edge in the EFSM. Since we always observe an edge and E is a previously unbound observer variable, $\text{is_edge}(E)$ will always succeed and bind E to a value. In q_1 we test for usage of the same state variable that was previously defined in q , and the existence of an observer edge. Similarly as in q , $\text{is_edge}(F)$ will always succeed and bind F to a value. On each transition in the observer, essential information is stored in the observer state, i.e., the name of the state variable, V , the edge where the state variable is defined, E , and the edge where the state variable is used, F . \square

3.2.1 Syntax for ERLANG/OBS

The ERLANG/OBS constructs are summarized in the grammar found in Table 2.1, describing the ERLANG syntax base definitions, and in Table 3.1, describing the included subset of, and extensions to, ERLANG expressions. Further, the syntax for observer predicate definitions is given in Table 3.2. In the following sections these constructs are further explained.

We assume specifications of observers to be well-formed such that types of input values and return values from functions are as expected when evaluated. ERLANG/OBS uses the same basic and compound values as ERLANG/EFSM, as defined in Table 2.1.

Observer variables

Observer variables, denoted u , share the syntax with, and are interpreted as, local variables in ERLANG/EFSM, see Section 2.3.1. I.e., an observer variable can only be bound to a value once and has a scope within the enclosing clause.

Observer expressions

Observer expressions oe are tuples, lists, and pattern matches. A tuple expression $\{oe_1, \dots, oe_n\}$ and a cons expression $[oe_1 | oe_2]$ is evaluated by first evaluating the sub-expressions oe_1 to oe_n .

The pattern match $cp=oe$ where cp is a pattern and oe an observer expression, binds unbound observer variables in cp to values, if pattern matching is possible. See Section 2.3.1 for a description of pattern matching.

Observer locations and observer state expressions

An *observer state* is of form $\iota(d_1, \dots, d_n)$ where ι is an *observer location*, and d_1, \dots, d_n are values, ranging over some domain. Examples of such domains include the set of locations, the set of edges, or the set of state variables of an ERLANG/EFSM specification. An *observer state expression* is of form

$$\iota(oe_1, \dots, oe_n)$$

$emop$	$::=$	$is_targetloc \mid is_sourceloc \mid is_definedvar \mid is_usedvar$ $\mid is_edge \mid is_startloc \mid is_stoploc \mid is_da$ $\mid is_sourceval \mid is_targetval \mid is_cc \mid is_mcc \mid is_mcdc$ $\mid is_eventtype \mid is_eventpars$	
f^{op}	$::=$	$emop \mid atom$	
ι	$::=$	$atom$	
ous	$::=$	$observer \mid filter \mid property$	
$obsge$	$::=$	$bv \mid u$ $\mid f^{op}(obsge_1, \dots, obsge_n)$ $n \geq 0$ $\mid guardop(obsge_1, \dots, obsge_n)$ $n > 0$	
h	$::=$	$obsge_1, \dots, obsge_n$ $n > 0$	
oe	$::=$	$bv \mid u \mid [oe_1 \mid oe_2] \mid \{oe_1, \dots, oe_n\} \mid (oe)$ $n > 0$ $\mid cp=oe$ $\mid guardop(oe_1, \dots, oe_n)$ $n > 0$ $\mid arithmetic(oe_1, \dots, oe_n)$ $n > 0$ $\mid \{next_state, \iota(oe_1, \dots, oe_n)\}$ $n \geq 0$	
oce	$::=$	oe_1, \dots, oe_n $n > 0$	
$obsedge$	$::=$	$\iota(u_1, \dots, u_m) \text{ when } h_1 \rightarrow oce_1;$ $m \geq 0$ \vdots $n > 0$ $\iota(u_1, \dots, u_m) \text{ when } h_n \rightarrow oce_n.$	
$olocdec$	$::=$	$-obs_locations(\iota_0, atlist_1, atlist_2).$	
$ousedec$	$::=$	$-obs_usage(\iota_0, ous).$	
$opredec$	$::=$	$-obs_predicates([f_1^{op}/i_1, \dots, f_n^{op}/i_n]).$ $n \geq 0$	
$obsdec$	$::=$	$ousedec \mid opredec$	
$obsdecs$	$::=$	$olocdec \ obsdec^*$	
$obspec$	$::=$	$obsdecs \ opdef^* \ obsedge^+$	

Table 3.1. ERLANG/OBS *syntax*.

where each oe_i is an observer expression, and is evaluated to an observer state by evaluating the observer expressions oe_1, \dots, oe_n to values. The observer locations that occur in an observer must be declared by

`-obs_locations({startloc, stoplocs, otherlocs}).`

where *startloc* is the name of an initial observer location, *stoplocs* is a non-empty list with the names of the accepting observer locations, i.e., observer locations from which there are no outgoing observer edges and *otherlocs* is a list with all observer locations that are neither an initial nor an accepting observer location. We use L^o to denote all observer locations in the **ERLANG/OBS** specification. For convenience, the initial observer location, *startloc*, is sometimes also used as the “name” of the observer and the accepting locations are “names” of coverage items.

Observer predicates

Observer predicates, represented by $f^{op}(obsge_1, \dots, obsge_n)$, are inspired by Prolog and optionally have one or more arguments. As a side effect, an observer predicate binds unbound observer variables, that occur in its arguments, to values. We assume a set of built-in observer predicates (*emop* in Table 3.1) that need no further declaration or definition. Observer predicates can also be user defined, and should then be declared by

`-obs_predicates([f1op/integer1, ..., fnop/integern]).`

How to define such declared user defined observer predicates is described in Section 3.3.

Observer guard expressions

Observer guard expressions *obsge* are boolean conditions, used as guards on observer edges. An observer guard expression must consist of a basic value, an observer variable, or the application of either an observer predicate (f^{op}) or a guard operator (*guardop*), to another observer guard expression. Note that comma “,” is a shorthand notation for the logical operator and.

Observer edges

The observer edges originating from an observer location are defined using a list of observer edge clauses of form

$$\begin{aligned} &\iota(u_1, \dots, u_m) \text{ when } h_1 \rightarrow oce_1; \\ &\quad \vdots \\ &\iota(u_1, \dots, u_m) \text{ when } h_n \rightarrow oce_n. \end{aligned}$$

where ι is the location from which the observer edges originate, and for each observer edge with index j , ranging over $1 \dots n$,

- u_1, \dots, u_m are observer variables and represent formal parameters to an observer state expression,
- h_j is an observer guard expression, and
- oce_j is an observer expression that must evaluate to a tuple of form

$$\{\text{next_state}, \iota'(d_1, \dots, d_k)\}$$

which represents a transition to the observer state $\iota'(d_1, \dots, d_k)$ (where ι' may be ι).

The location ι must be declared as an observer location. In an observer expression oce_i each observer variable must be bound to a value before its use.

Observer usage

In this thesis, we mostly use observers to define coverage items and coverage criteria. However, by viewing observers as acceptors of runs of **ER-LANG**/EFSM specifications, they can also be used for other purposes. The intended usage of an observer is declared by an observer usage declaration of form

$$\text{-obs_usage}(\iota_0, \text{ous})$$

where ι_0 is an initial observer location, and *ous* is an observer usage declaration. This is an atom, which can be one of the following:

- **observer** means that the observer defines a set of coverage items; in this case the test generation algorithm will include a new test case in a test suite whenever a new accepting observer location is reached,
- **filter** means that the observer defines a criterion for including runs in a test suite; in this case all test cases that cause an accepting observer location to be reached should be included,
- **property** means that the observer defines a correctness property; it can be used by a tool to check that no run satisfies the property represented by the observer; note that this usage is not intended for test suite generation.

If no usage declaration is provided, **observer** usage is assumed. The other usages of observers are further discussed in Section 5.5.

3.3 Defining observer predicates

Observer predicates represent tests that depend on components of the current annotated computation step. For the definition of observer pred-

icates we define a set of match variables, representing annotated computation steps. An observer predicate may then perform tests on the current values of match variables and provided observer variables. Observer predicates are inspired by predicates in Prolog, in that their evaluation will bind previously unbound variables to values in the local environment. We assume a set of built-in observer predicates (*emop* in Table 3.1) that need no further declaration or definition. Additionally, observer predicates may be user defined after they have been declared, see Section 3.2.1. In the following we describe how declared user defined observer predicates are defined.

3.3.1 Match variables

Recall that observers monitor EFSM runs, i.e., sequences of computation steps. Each computation step in such a sequence is represented as a valuation of *match variables*. Match variables are bound to different components of an annotated computation step. In this thesis, we use the match variables shown in Figure 3.4, together with their intended meaning.

3.3.2 Match functions

Match functions are introduced to be able to express more observer predicate definitions. Thus, as observer predicates, match functions operate on match variables and the formal parameters u_1, \dots, u_n to observer predicates. The match functions, used to define observer predicates in this thesis, are found in Figure 3.5. However, we will assume that in general match functions can be defined freely and allow for definitions with any valid **ERLANG** expression. Match functions are discussed in more detail in Section 3.5.2.

3.3.3 Observer predicate definition in **ERLANG/OBS**

The syntax for definition of observer predicates is found in Table 2.1, Table 2.2 and Table 3.2. An observer predicate is defined using a clause

$$f^{op}(u_1, \dots, u_n) \rightarrow ope.$$

where f^{op} is the name of the observer predicate, u_1, \dots, u_n are observer variables and represent formal parameters to the observer predicate, and ope is a boolean expression over values and match variables, pattern matches ($cp=ope$), guard operators ($guardop(ope_1, \dots, ope_n)$), and match functions ($mf(ope_1, \dots, ope_n)$).

Match variables for computation steps: $\langle l, \sigma \rangle \xrightarrow{a(\bar{d})/\bar{b}} \langle l', \sigma' \rangle$	
Eventtype	is the event type a of the occurring input event,
Eventpars	is the sequence \bar{d} of parameters of the occurring input event,
Outevents	is the sequence of occurring output events \bar{b} ,
SourceLoc	is the location l just before the computation step,
SourceEnv	is the environment σ just before the computation step,
TargetLoc	is the location l' just after the computation step,
TargetEnv	is the environment σ' just after the computation step.
Match variables for ERLANG/EFsm edge clauses:	
$l(a, u_1, \dots, u_m) \text{ when } g \rightarrow \overline{v = \bar{e}}, \bar{b}, \{\text{next_state}, l'\}$	
Edgevars	is the local variables u_1, \dots, u_n of the ERLANG/EFsm edge clause,
Guard	is the guard expression g of the ERLANG/EFsm edge clause,
Assigns	is the sequence of bindings $\overline{v = \bar{e}}$ of the ERLANG/EFsm edge clause,
Outexprs	is the sequence of output expressions \bar{b} of the ERLANG/EFsm edge clause.

Figure 3.4. Match variables accessible for an observer.

mf	$::=$	member lookup map eval subst	
		affect lhs rhs vars conds to_bool	
mv	$::=$	Eventtype Edgevars Eventpars	
		SourceLoc TargetLoc Guard Assigns	
		Outevents Outexprs SourceEnv TargetEnv	
u	$::=$	$bv \mid u$	
ope	$::=$	$bv \mid u \mid mv \mid [ope_1 \mid ope_2] \mid \{ope_1, \dots, ope_n\}$	$n > 0$
		$cp = ope$	
		$guardop(ope_1, \dots, ope_n)$	$n > 0$
		$mf(ope_1, \dots, ope_n)$	$n \geq 0$
$opdef$	$::=$	$f^{op}(u_1, \dots, u_n) \rightarrow ope.$	$n \geq 0$

Table 3.2. ERLANG/OBS observer predicate definition syntax.

member
 is a predicate such that `member(Element, List)` returns `true` if *Element* is a member of the list *List* and `false` otherwise.

lookup
 searches a list of tuples (key/value list), such that `lookup(Key, KeyValList)` returns the 2:nd element in the first tuple in *KeyValList* whose 1:st element matches *Key*. If no such tuple is found, `false` is returned.

map
 is a function such that `map(Fun, List)` applies the function *Fun* on each element in *List* and returns the resulting list. Additional arguments can be passed to *Fun* by supplying a list with these arguments in a tuple with *Fun*, i.e., `map({Fun, Args}, List)`. In this case all additional arguments *Args* will be passed *after* the single element from *List*, i.e., `Fun(x, a1, a2, a3)` where *x* is a member of *List* and *Args* is [*a*₁, *a*₂, *a*₃].

eval
 is a function such that `eval(GE, Env)` evaluates *GE* in an environment *Env*.

subst
 is a function such that `subst(GE, R, Bool)` substitutes all occurrences of a sub-expression *R* in a guard expression *GE* with *Bool*.

affect
 is a function such that `affect(Assign, Var1, Var2)` returns *Assign* if it is a binding of *Var*₂ to some expression that includes *Var*₁, otherwise the empty list is returned.

lhs
 is a function such that `lhs(Assign)` returns the left hand side expression of a binding *Assign*. A left hand side expressions is always assumed to be a state variable.

rhs
 is a function such that `rhs(Assign)` returns the right hand side expression of a binding.

vars
 is a function such that `vars(Exp)` returns a list with all state variables found in *Exp* where *Exp* is a list or function expressions.

conds
 is a function such that `conds(Guard)` returns a list of all conditions (relational operators, *relop* and type tests, *typetest*), in lexical order, in a boolean expression *Guard*.

to_bool
 is a function that maps anything that is not the atom `false` to `true`.

Figure 3.5. Match functions, for defining observer predicates in ERLANG/OBS, used in this thesis.

Example 3.3 We can define an observer predicate `is_sourceval/2`, which returns `true` if its first argument is an `ERLANG/EFSM` state variable, and its second argument is the value of this state variable just before the occurrence of a computation step. `is_sourceval/2` can be defined as

```
is_sourceval(V,Val) ->
    to_bool(Val=lookup(V,SourceEnv)).
```

If `is_sourceval/2` is invoked in a call of form `is_sourceval(X,Y)`, where X is an observer variable that is bound to the state variable v , and Y is an unbound observer variable, then according to the above definition, the match function `lookup` will return the value of v in `SourceEnv`. If v is defined in `SourceEnv` then by a pattern match, the return value from `lookup` is bound to Y . The match function `to_bool` then maps any return value from the pattern match that is not the atom `false` to `true`. If v is not defined in `SourceEnv` then the return value from `lookup` is `false`. Thus, Y is bound to `false` in this case.

Note that if Y instead is a bound observer variable and the pattern match fails, `false` will be returned. This behavior is different from `ERLANG/EFSM` (and standard `ERLANG`, where a failed pattern match raises an exception). Also note that if both X and Y are unbound observer variables then X will be bound to some state variable and Y to its value before the computation step. Note that `is_targetval/2` is defined similarly, but instead utilizing the `TargetEnv` match variable. \square

Example 3.4 An observer predicate `is_definedvar/1` returns `true` if its argument is an `ERLANG/EFSM` state variable bound to a value in an assignment in the computation step. This can be expressed using match functions `member`, `map` and `lhs`, and the match variable `Assigns` as

```
is_definedvar(V) ->
    member(V,map(lhs,Assigns)).
```

If `is_definedvar/1` is invoked in a call of form `is_definedvar(X)` where X is an observer variable that is bound to the state variable v , then according to the above definition, the match function `member` returns `true` if X is a member of the list created by taking the left-hand-side (`lhs`) of each binding in the match variable `Assigns`. If X instead is an *unbound* observer variable, `member` will bind X to the name of *some* `ERLANG/EFSM` state variable v for which there exists an assignment $v = e$ in the match variable `Assigns`. If there exists no binding of X in `Assigns` the observer predicate returns `false` and X is bound to the atom `undefined`. \square

For each match variable in Figure 3.4

Observer Predicate	Match variable
is_sourceloc/1	SourceLoc
is_targetloc/1	TargetLoc
is_eventtype/2	Eventtype
is_eventpars/1	Eventpars

Table 3.3. *Observer predicates defined by pattern matching with a match variable.*

we can define an observer predicate that is true whenever the match variable matches a given value. For example, for the match variable `TargetLoc` we define an observer predicate `is_targetloc/1` with a single observer argument to be true only when the target location of the occurring edge matches the argument. The `is_targetloc/1` observer predicate definition can be expressed in `ERLANG/OBS` as

```
is_targetloc(L) ->
    to_bool(L=TargetLoc).
```

Several of the observer predicates in Table 3.1 can be defined similarly by pattern matching the argument against a corresponding match variable. See Table 3.3 for a complete list.

Slightly more complicated observer predicates can be defined by assuming certain values on match variables or combining several match variables. Below, we give a few more examples of observer predicate definitions. In Section 4 we will continue with more observer predicate definitions, and give more examples on how observer predicates can be used to specify coverage criteria.

`is_startloc()`

Returns `true` if the current value of the match variable `SourceLoc` is the initial location of the EFSM, and `false` otherwise. The observer predicate is defined as

```
is_startloc() ->
    to_bool( $l_0$  = SourceLoc).
```

where l_0 is the initial location in the EFSM we are monitoring.

`is_stoploc()`

Returns `true` if the current value of the match variable `TargetLoc` is a stop location, i.e., an annotated location in which there are no outgoing edges, and `false` otherwise. The observer predicate is defined as

```
is_stoploc() ->
  member(TargetLoc, Stoplocs).
```

where `Stoplocs` is a list with the names of the stop locations L^{stop} in the EFSM we are monitoring.

`is_edge(E)`

Returns `true` if `E` is a tuple $\{l_{i-1}, a_i, \overline{u_i}, g_i, l_i\}$, uniquely identifying a single edge clause

$$l_{i-1}(a_i, u_{i1}, \dots, u_{im}) \text{ when } g_i \rightarrow \overline{v_i} = \overline{c_i}, \overline{b_i}, \{\text{next_state}, l_i\}$$

in the EFSM from the current annotated computation step, and the current values of the match variables are as follows: `SourceLoc` is l_{i-1} , `Eventtype` is a_i , `Edgevars` is u_{i1}, \dots, u_{im} , `Guard` is g_i , and `TargetLoc` is l_i . The observer predicate is defined in `ERLANG/OBS` as

```
is_edge({SL, ET, EV, G, TL}) ->
  to_bool({SL, ET, EV, G, TL} =
    {SourceLoc, Eventtype, Edgevars, Guard, TargetLoc}).
```

3.4 Graphical observer notation

To visualize observers, we introduce a graphical notation for observers. We use the symbol \bullet to represent the initial observer location and the symbol \odot to represent accepting observer locations. An observer guard expression is written in association to the observer edge to which it belongs. The graphical observer notation uses the same syntax for observer guard expressions and observer states as `ERLANG/OBS`, defined in Section 3.2.1. It can be noted the graphical representation cannot represent an observer edge when the source and target observer state share the same observer location but the observer states are not identical. e.g., because an observer variable is a list that is bound to a new value by the observer edge. See examples of such observers in Section 4.1.3.

Example 3.5 In Figure 3.6 two similar observers for definition-use pairs are shown, both with an accepting observer location du . Observer (i) represent a *single* coverage item in $du(Y', e_1, e_2)$ for an `ERLANG/EFM` state variable with a name Y , defined on edge e_1 and used on edge e_2 . The state variable must not be redefined in the run between these two

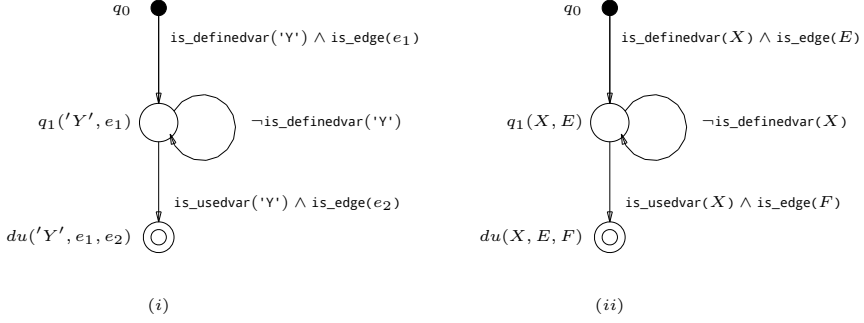


Figure 3.6. Examples of (i) observer monitoring definition (on edge e_1) and use (on edge e_2) of state variable Y , (ii) observer monitoring definition-use for *any* triples of edges and state variables.

edges. Here e_i is a short hand notation for a tuple $\{l_{i-1}, a_i, \overline{u_i}, g_i, l_i\}$, uniquely identifying a single edge clause in the EFSM.

Figure 3.6(ii) shows an observer that specifies definition-use pair coverage for *any* state variable, and pair of edges in the EFSM. Thus, $du(V, E, F)$ represent a coverage item when

- the observer variable V is bound to a name of a state variable in the EFSM,
- the observer variable E is bound to the edge in the EFSM on which the V state variable was bound, and
- the observer variable F is bound to the edge in the EFSM on which the V state variable was used.

An **ERLANG/OBS** specification of this observer is found in Example 3.2. A definition of the observer predicate **is_edge/1** was given in Section 3.3. The observer predicates **is_definedvar/1** and **is_usedvar/1** are further defined in Section 4.1.4. \square

3.5 Operational semantics of **ERLANG/OBS**

In this section we give the operational semantics for **ERLANG/OBS** in the form of transition rules between structural states. In Section 2.4 the operational semantics for **ERLANG/EFSM** was given. To simplify the presentation we will, as much as possible, reuse **ERLANG/EFSM** operational semantics, including structure of presentation, and definitions of structural states and transition rules.

Recall that in **ERLANG/EFSM**, a structural state is defined to be a tuple of form $\langle e, \overline{b}, \sigma, \rho \rangle$, where σ is global environment with bindings of state variables. Since an observer has (local) observer variables, but no state variables, we will replace the third component of the structural state by a

global environment which holds the current values of the match variables. Thus, the global environment can be thought of as representing each annotated computation step of the EFSM, using match variables.

Definition 3.6 Let D be the domain of values, and MV be the set of match variables. A *structural observer state* is a tuple

$$\langle oe, \epsilon, \varsigma, \rho \rangle$$

where

- oe is an **ERLANG/OBS** expression,
- ϵ is an empty sequence of output events; this component is never used, we leave it here for compatibility with structural states in the semantics for **ERLANG/EFSM**,
- $\varsigma \in MV \rightarrow D$ is a global environment with bindings to the match variables MV , and
- $\rho \in U \rightarrow D$ is a local environment with bindings to the observer variables U , □

We will sometimes use “structural state” for “structural observer state”.

Recall, from Section 2.4, that an empty environment is denoted \bullet and an empty sequence of output events is denoted ϵ . An *initial structural state* is the tuple

$$\langle \iota_0(), \epsilon, \varsigma, \bullet \rangle$$

where $\iota_0()$ is an observer state and ς holds the bindings of match variables to expressions from the current annotated computation step. A transition between two structural states

$$\langle oe, \varsigma, \rho \rangle \Rightarrow \langle d, \epsilon, \varsigma, \rho' \rangle$$

denotes that oe can be evaluated in a global environment ς and local environment ρ to a value d . Since this evaluation is performed in the context of one annotated computation step of the EFSM, the global environment ς does not change. In **ERLANG/OBS** (as in **ERLANG/EFSM**) pattern matching is used to bind variables to values. For **ERLANG/OBS** we reuse the match relation from Section 2.4.1.

3.5.1 Observer expressions

All expressions in **ERLANG/OBS** can be evaluated to **ERLANG/EFSM** values. In this section we outline the transition rules for how this is accomplished. For **ERLANG/EFSM** values, tuples, conses, logical operators, arithmetic expressions, observer variable definition and usage, and sequences of observer expressions oce , we use the transition rules from Section 2.4. Observer guard expressions compute boolean expressions of observer predicates, and occur in observer edge clauses.

Similarly as in **ERLANG/EFM**S in Section 2.4.2, observer variables are also bound to values by pattern matching in observer expressions. Note that, in addition, observer variables are also bound to values by observer predicates, if not previously bound. Observer variables must be bound in the local environment, ρ , to a value before they are used.

3.5.2 Observer predicates

Observer predicates require certain attention and a number of new transition rules since they do not occur in **ERLANG/EFM**S. The grammar is found in Table 3.2. An observer predicate is defined by a clause

$$f^{op}(u_1, \dots, u_n) \rightarrow ope.$$

where f^{op} is the name of the observer predicate, u_1, \dots, u_n are observer variables and represent formal parameters to the observer predicate, and ope (observer predicate expression) is a *boolean* expression built from expressions of form

- values, observer variables, match variables, tuples and conses,
- $cp=ope$, i.e., matching of a pattern cp with an expression ope ,
- $guardop(ope_1, \dots, ope_n)$, i.e., application of a guard operator over expressions ope_1, \dots, ope_n , see **ERLANG/EFM**S syntax, Table 2.1, and
- $mf(ope_1, \dots, ope_n)$, i.e., application of a match function over expressions ope_1, \dots, ope_n , see Section 3.5.2.

A call to an observer predicate $f^{op}(obsge_1, \dots, obsge_n)$ is evaluated such that previously unbound variables are bound to **ERLANG/EFM**S values in the local environment. The transition rule is

$$\frac{\langle ope[u_1, \dots, u_n \mapsto obsge_1, \dots, obsge_n], \varsigma, \rho \rangle \Rightarrow \langle bool, \epsilon, \varsigma, \rho' \rangle}{\langle f^{op}(obsge_1, \dots, obsge_n), \varsigma, \rho \rangle \Rightarrow \langle bool, \epsilon, \varsigma, \rho' \rangle}$$

where the resulting local environment ρ' includes bindings to all unbound observer variables in $obsge_1, \dots, obsge_n$. How observer predicates bind unbound observer variables to **ERLANG/EFM**S values, is reflected in the operational semantics for **ERLANG/OBS** by the transition rules for pattern matching and match functions. For example, the observer predicate `is_targetloc/1` is defined by the clause

```
is_targetloc(Loc) ->
  to_bool(Loc=TargetLoc).
```

Thus, the transition rule that can be applied, for a call `is_targetloc(L)` in an observer guard after substitution, is

$$\frac{\langle to_bool(L=TargetLoc), \varsigma, \rho \rangle \Rightarrow \langle bool, \epsilon, \varsigma, \rho' \rangle}{\langle is_targetloc(L), \varsigma, \rho \rangle \Rightarrow \langle bool, \epsilon, \varsigma, \rho' \rangle}$$

where the pattern match $L = \text{TargetLoc}$ matches only if (1) L is an observer variable, bound to a value in ρ , that match the value of TargetLoc bound to a value in ς , or (2) if L is a previously unbound observer variable. The match operator $=$ (slightly modified from **ERLANG/EFSM**, see below) returns the matching pattern (i.e., value of TargetLoc) and we therefore use the match function `to_bool` to translate the result to a boolean. Thus, `is_targetloc/1` will return `true` if a pattern match is possible and `false` if not.

Pattern matching

The pattern match $cp = ope$ where cp is a pattern and ope an observer predicate expression, binds any unbound observer variable in cp to a value, if pattern matching is possible. The match operator in **ERLANG/EFSM**, see Section 2.4.1, is only defined when a match succeeds between the two operands (and is not allowed in guards). However, in **ERLANG/OBS** we use the match operator in observer predicates (that occur in guards) and need to additionally consider the case when a match fails. A failing match implies a conflict between two expressions with bound variables. We therefore define an additional transition rule for pattern matching when all observer variables are bound to values and the pattern match fails, as follows

$$\frac{\langle cp == ope, \varsigma, \rho \rangle \Rightarrow \langle \text{false}, \epsilon, \varsigma, \rho \rangle}{\langle cp = ope, \varsigma, \rho \rangle \Rightarrow \langle \text{false}, \epsilon, \varsigma, \rho \rangle}$$

where the transition rule for the relational operator $==$ (from **ERLANG/EFSM**, see Section 2.7.2) is only defined when all its operands are bound to values. Thus, a failing pattern match only has a transition when there is no unbound variable in cp .

Match variables

Match variables are given values at the occurrence of an annotated computation step. To proceed with more precise definitions of observers, used in this thesis, we make a few assumptions on underlying representations of values of match variables. The actual representation is tool implementation dependent. Given that we observe a finite state machine, we assume all match variables have finite domains. Let values be represented by themselves and all expressions be represented by records. For example, a record

`#variable{name=V}`

represent a variable V , and a record

`#func{op=F, args=Args}.`

Match variables for computation steps: $\langle l, \sigma \rangle \xrightarrow{a(\bar{d})/\bar{b}} \langle l', \sigma' \rangle$	
Eventtype	<i>atom</i>
Eventpars	$[d_1, \dots, d_n]$
Outevents	$[\text{\#func}\{\text{op}=b_1, \text{args}=[d_{11}, \dots, d_{1m}]\}$
	\vdots
	$\text{\#func}\{\text{op}=b_n, \text{args}=[d_{n1}, \dots, d_{nm}]\}]$
SourceLoc	<i>atom</i> ,
SourceEnv	$[\{v_1, d_1\} \cdots \{v_n, d_n\}]$
TargetLoc	<i>atom</i> ,
TargetEnv	$[\{v_1, d_1\} \cdots \{v_n, d_n\}]$
Match variables for ERLANG/EFSM edge clauses:	
$l(a, u_1, \dots, u_m) \text{ when } g \rightarrow \overline{v} \equiv \bar{\epsilon}, \bar{b}, \{\text{next_state}, l'\}$	
Edgevars	$[u_1, \dots, u_m]$
Guard	$\text{\#func}\{\text{op}=\textit{guardop}, \text{args}=[ge_1, \dots, ge_n]\}$
Assigns	$[\{v_1, e_1\} \cdots \{v_n, e_n\}]$
Outexprs	$[\text{\#func}\{\text{op}=b_1, \text{args}=[e_{m1}, \dots, e_{m1}]\}$
	\vdots
	$\text{\#func}\{\text{op}=b_n, \text{args}=[e_{m1}, \dots, e_{mn}]\}]$

Figure 3.7. Representation of the match variables in Figure 3.4 as ERLANG/EFSM expressions.

represent a function call to F with arguments $Args$, i.e., $F(Args)$. Binding of a state variable is represented by a tuple $\{V, E\}$, where V is a state variable and E an ERLANG/EFSM expression represented by records as above.

Assigns, SourceEnv and TargetEnv

The match variables for assignments, source and target environment are all represented by a list of tuples, $\{V, E\}$ i.e., the same form as state variable bindings above. Further, in a computation step σ and σ' always bind state variables to values, i.e., in the representation of SourceEnv and TargetEnv, E is always an ERLANG/EFSM value.

Eventpars and Edgevars

The match variable for input event parameters is represented by a list of ERLANG/EFSM values. The match variable for input expression parameters is represented by a list of variable names.

Outevents and Outexprs

The match variables for output events and output expressions are both represented by a list of function records, i.e., $\#func\{op=F, args=Args\}$ records where F is an output event type, and $Args$ a list of arguments. For Outevents the arguments are ERLANG/EFSM values and for Outexprs the arguments are ERLANG/EFSM expressions.

Guard

The match variable for a guard is a record $\#func\{op=F, args=Args\}$ where F is an atom representing one of the ERLANG/EFSM guard operators (*guardop* in Table 2.1) and $Args$ is a list $[ge_1, \dots, ge_n]$ where each guard expression ge_k is a value d , a variable, represented by a record $\#variable\{name=V\}$ for some variable V , a configuration access function or a guard operator, both represented by a record of form $\#func\{op=F, args=Args\}$.

Figure 3.7 summarizes the representation of the match variables assumed in this thesis for non-symbolic execution. Match variables for symbolic execution is represented similarly where each symbolic parameter V is represented with a record $\#variable\{name=V\}$.

Match functions

Match functions are used to help define observer predicates. In this section we define a set of match functions necessary to define the observer predicates in this thesis. For the definition of match functions we need additional transition rules. However, we will assume that in general match functions can be defined freely and allow for definitions with *any* valid ERLANG expression.

In Figure 3.7 we give an assumed representation of match variables in terms of **ERLANG/EFMS** expressions. This is the representation assumed in the definitions of the match functions. We distinguish between binding and non-binding match functions. For match functions that bind any unbound observer variable to a value and appear in the thesis, all the necessary transition rules are given below. For match functions that assume all observer variables used are already bound to values, transitions rules are straightforward and we omit them here.

Binding match functions

We begin by describing match functions that bind unbound observer variables to **ERLANG/EFMS** values.

member(*Element*, *List*)

A predicate such that **member**(*Element*, *List*) returns true if *Element* is a member of the list *List* (with length $n \geq 0$) and false otherwise. The transition rules for **member** are as follows.

$$\frac{\begin{array}{l} \langle e_1, \varsigma, \rho \rangle \Rightarrow \langle d, \epsilon, \varsigma, \rho \rangle \\ \langle e_2, \varsigma, \rho \rangle \Rightarrow \langle [d_1, \dots, d_n], \epsilon, \varsigma, \rho \rangle \\ \exists i : 1 \leq i \leq n :: \text{match}(d, d_i, \rho, \rho) \end{array}}{\langle \text{member}(e_1, e_2), \varsigma, \rho \rangle \Rightarrow \langle \text{true}, \epsilon, \varsigma, \rho \rangle}$$

if e_1 is a value or an expression that evaluates to a bound observer variable, which is a member of a list e_2 ,

$$\frac{\begin{array}{l} \langle e_1, \varsigma, \rho \rangle \Rightarrow \langle d, \epsilon, \varsigma, \rho \rangle \\ \langle e_2, \varsigma, \rho \rangle \Rightarrow \langle [d_1, \dots, d_n], \epsilon, \varsigma, \rho \rangle \\ \forall i, 1 \leq i \leq n : \neg \text{match}(d, d_i, \rho, \rho) \end{array}}{\langle \text{member}(e_1, e_2), \varsigma, \rho \rangle \Rightarrow \langle \text{false}, \epsilon, \varsigma, \rho \rangle}$$

if e_1 is a value or an expression that evaluates to a bound observer variable, which is *not* a member of a list e_2 , If $n = 0$ an empty list is implied.

$$\frac{\begin{array}{l} \langle e_1, \varsigma, \rho \rangle \Rightarrow \langle u, \epsilon, \varsigma, \rho \rangle \\ u \text{ is an observer variable, not defined in } \rho \\ \langle e_2, \varsigma, \rho \rangle \Rightarrow \langle [], \epsilon, \varsigma, \rho \rangle \end{array}}{\langle \text{member}(e_1, e_2), \varsigma, \rho \rangle \Rightarrow \langle \text{false}, \epsilon, \varsigma, \rho[u \mapsto \text{undefined}] \rangle}$$

if e_1 is an expression that evaluates to an unbound observer variable, e_2 is the empty list, and

$$\frac{\begin{array}{l} \langle e_1, \varsigma, \rho \rangle \Rightarrow \langle u, \epsilon, \varsigma, \rho \rangle \\ u \text{ is an observer variable, not defined in } \rho \\ \langle e_2, \varsigma, \rho \rangle \Rightarrow \langle [d_1, \dots, d_n], \epsilon, \varsigma, \rho \rangle \end{array}}{\langle \text{member}(e_1, e_2), \varsigma, \rho \rangle \Rightarrow \langle \text{true}, \epsilon, \varsigma, \rho[u \mapsto d_i] \rangle}$$

if e_1 is an expression that evaluates to an unbound observer variable and e_2 is a non-empty list. In this case *any* member of e_2 is a possible solution. Note that we do not define any transition rule for the case when e_2 is an unbound observer variable as that would generate an infinite number of possible solutions.

Example 3.7 The observer predicate `is_usedvar/1` (see also Section 4.1.4) is true if a (state) variable v is used in an `ERLANG/EFsm` edge clause, i.e., found in match variable `Guard`, `Assigns` or `Outexprs`. We define the observer predicate using the match functions as

```
is_usedvar(U) ->
    member(U, vars(map(rhs,Assigns)) ++
              vars(Guard) ++
              vars(Outexprs) ).
```

This means that if `U` is evaluated to a value it must match the name of a state variable in the current `ERLANG/EFsm` edge clause, and if `U` is evaluated to a previously unbound variable it is bound to the variable name of one of (the possibly many) variables used in `Assigns`, `Guard` and `Outexprs`. The operator `++` is `ERLANG` notation for appending two lists. The match functions `vars`, `map`, and `rhs` are defined below. \square

`lookup(Key, KeyValList)`

Searches a list of tuples, `KeyValList` (with length $n \geq 0$), for a tuple whose 1:st element matches `Key`. If found, the 2:nd element of such a tuple is returned, otherwise the boolean `false` is returned. The transition rules for `lookup` are as follows.

$$\frac{\begin{array}{l} \langle e_1, \varsigma, \rho \rangle \Rightarrow \langle d_1, \epsilon, \varsigma, \rho \rangle \\ \langle e_2, \varsigma, \rho \rangle \Rightarrow \langle [\{d_{11}, d_{12}\}, \dots, \{d_{n1}, d_{n2}\}], \epsilon, \varsigma, \rho \rangle \\ \forall j, 1 \leq j < i : \neg \text{match}(d_1, d_{j1}, \rho, \rho) \\ \text{match}(d_1, d_{i1}, \rho, \rho) \end{array}}{\langle \text{lookup}(e_1, e_2), \varsigma, \rho \rangle \Rightarrow \langle d_{i2}, \epsilon, \varsigma, \rho \rangle}$$

if e_1 is a value or a bound observer variable, which is a key in the key/value list e_2 ,

$$\frac{\begin{array}{l} \langle e_1, \varsigma, \rho \rangle \Rightarrow \langle d_1, \epsilon, \varsigma, \rho \rangle \\ \langle e_2, \varsigma, \rho \rangle \Rightarrow \langle [\{d_{11}, d_{12}\}, \dots, \{d_{n1}, d_{n2}\}], \epsilon, \varsigma, \rho \rangle \\ \forall j, 1 \leq j \leq n : \neg \text{match}(d_1, d_{j1}, \rho, \rho) \end{array}}{\langle \text{lookup}(e_1, e_2), \varsigma, \rho \rangle \Rightarrow \langle \text{false}, \epsilon, \varsigma, \rho \rangle}$$

if e_1 is a value or a bound observer variable, which is *not* a key in the key/value list e_2 ($n \geq 0$), and

$$\frac{\begin{array}{l} \langle e_1, \varsigma, \rho \rangle \Rightarrow \langle u, \epsilon, \varsigma, \rho \rangle \\ u \text{ is an observer variable, not defined in } \rho \\ \langle e_2, \varsigma, \rho \rangle \Rightarrow \langle [\{d_{11}, d_{12}\}, \dots, \{d_{n1}, d_{n2}\}], \epsilon, \varsigma, \rho \rangle \end{array}}{\langle \text{lookup}(e_1, e_2), \varsigma, \rho \rangle \Rightarrow \langle d_{i2}, \epsilon, \varsigma, \rho[u \mapsto d_{i1}] \rangle}$$

if e_1 is an expression that evaluates to an unbound observer variable and e_2 is a non-empty key/value list. In this case *any* tuple in e_2 is a possible solution. If e_2 evaluates to an empty list ($n = 0$), u is bound to **undefined** and **false** returned. Note that we do not define any transition rule for the case when e_2 is an unbound observer variable as that would generate an infinite number of possible solutions.

Non-binding match functions

A number of match function do not bind observer variables to values. Essentially, these are user defined **ERLANG/EFM** functions where we allow some additional **ERLANG** expressions. The definition of the missing transition rules for these expressions are straightforward. Given that **lists** is an **ERLANG** module that implements the functions **sort/1** and **reverse/1**, (e.g., the standard **ERLANG/OTP** module). The additional **ERLANG** expressions that we use in the definition of non-binding match functions below are:

$L1++L2$ where $L1$ and $L2$ are lists and $++$ the **ERLANG** short hand notation for appending list $L2$ after $L1$.

element(N, T) where N is an integer and T is a tuple, **element** is an **ERLANG** function that returns the value of element N in the tuple T .

tuple_to_list(T) where T is a tuple and **tuple_to_list** is an **ERLANG** function that transforms T to a list.

apply($Fun, Args$) where Fun is an atom holding the name of a function and $Args$ is a list with arguments that is passed as arguments to Fun .

lists:sort($List$) where $List$ is a list, returns a sorted list.

lists:reverse($List$) where $List$ is a list, returns a reversed list.

Below we give **ERLANG** definitions, for all non-binding match functions, used in this thesis,

map is a function such that `map(Fun, List)` applies the function, *Fun* on each element in *List* and returns the resulting list. Additional arguments can be passed to *Fun* by supplying a list with these arguments in a tuple with *Fun*, i.e., `map({Fun, Args}, List)`. These additional arguments will be passed *after* the element from *List*. This can also be expressed as:

```
map(Fun,List) ->
    map2(Fun,List,[]).

map2(Fun,[],Out) ->
    Out;
map2({Fun,Args},[H|Rest],Out) ->
    map2(Rest,Out++[apply(Fun,[H|Args])]).
map2(Fun,[H|Rest],Out) ->
    map2(Rest,Out++[apply(Fun,[H])]).
```

eval is a function such that `eval(GE, Env)` evaluates *GE* in an environment *Env*. Note that a state variable must be bound in the environment, before usage. This can also be expressed as:

```
eval(#variable{name=V},Env) ->
    lookup(V,Env);
eval(G=#func{args=Args},Env) ->
    NewArgs=eval2(Args,Env,[]),
    G#func{args=NewArgs};
eval(G,Env) ->
    G.

eval2([],Env,NewArgs) ->
    lists:reverse(NewArgs);
eval2([G|Rest],Env,NewArgs) ->
    NewA=eval(G,Env),
    eval2(Rest,Env,[NewA|NewArgs]).
```

subst is a function such that `subst(GE, R, Bool)` substitutes all occurrences of a subexpression *R* in a guard expression *GE* with *Bool*. This can also be expressed as:

```
subst(R,R,Bool) ->
    Bool;
subst(G=#func{op=Op},R,Bool)
    when Op=='/=';Op=='=';Op=='<';
        Op=='>';Op=='<';Op=='>';Op=='is_integer';
        Op=='is_boolean';Op=='is_record';
```

```

        Op=='is_tuple';Op=='is_list' ->
    G;
subst(Guard=#func{args=Args},R,Bool) ->
    NewArgs=subst2(Args,R,Bool,[]),
    Guard#func{args=NewArgs},R,Bool).

subst2([],R,Bool,NewArgs) ->
    lists:reverse(NewArgs);
subst2([G|Rest],R,Bool,NewArgs) ->
    NewG=subst(G,R,Bool),
    subst2(Rest,R,Bool,[NewG|NewArgs]).

```

affect is a function such that *affect*(*Assign*, *Var*₁, *Var*₂) returns *Assign* if it is a binding of *Var*₂ to some expression that includes *Var*₁, otherwise the empty list is returned. A binding, as *Assign*, is a tuple {*K*, *V*} where *K* is a state variable and *V* an ERLANG/EFM expression. This can also be expressed as:

```

if
    member(Var1,vars([rhs(Assign)])) and
    (Var2==lhs(Assign)) -> Assign;
true -> []
end

```

lhs is a function such that *lhs*(*Assign*) returns the left hand side expression of a binding *Assign*. A left hand side expressions is always assumed to be a state variable.

```

element(1,Assign)

```

rhs is a function such that *rhs*(*Assign*) returns the right hand side expression of a binding *Assign* and can be expressed as

```

element(2,Assign)

```

vars is a function such that *vars*(*Exp*) returns a list with all state variables found in *Exp* where *Exp* is a list of ERLANG/EFM normal form expressions.

```

vars([]) -> [];
vars([Expr|Rest]) -> vars_in_expr(Expr)++vars(Rest);
vars(#func{args=Args}) -> vars(Args).

```

```

vars_in_expr(#func{args=Args}) ->
    vars(Args);
vars_in_expr(E) when is_tuple(E) ->
    vars(tuple_to_list(E));
vars_in_expr(E) when is_list(E) ->
    vars(E);
vars_in_expr(#variable{name=V}) -> [V];
vars_in_expr(_) -> [].

```

`conds` is a function such that `conds(Guard)` returns a list of all predicates (relational operators, *relop* and type tests, *typetest*), in lexical order, in a boolean expression *Guard*.

```
conds(A=#func{op=Op})
  when Op=='!=';Op=='=';Op=='<';
      Op=='>';Op=='<';Op=='>';Op=='is_integer';
      Op=='is_boolean';Op=='is_record';
      Op=='is_tuple';Op=='is_list' ->
    [A];
conds(#func{args=Args}) ->
  lists:sort(conds2(Args,[]));
conds(_) ->
  [].

conds2([].Conds) ->
  Conds;
conds2([A|Rest],Conds) ->
  conds2(Rest,Conds++conds(A)).
```

`to_bool` is a function that maps anything that is not the atom `false` to `true`.

```
to_bool(false) -> false;
to_bool(_) -> true.
```

3.5.3 Observer edge clauses

The transitions from an observer state $\iota(d_1, \dots, d_m)$ are defined by observer edge clauses

$$\begin{aligned} &\iota(u_1, \dots, u_m) \text{ when } h_1 \rightarrow oce_1; \\ &\quad \vdots \\ &\iota(u_1, \dots, u_m) \text{ when } h_n \rightarrow oce_n. \end{aligned}$$

Each of these clauses are evaluated by first matching the values d_1, \dots, d_m with the observer variables u_1, \dots, u_m , and if the match succeeds the observer guard expression h_k is evaluated. For *some* clause k with all observer variables, u_1, \dots, u_m matching and observer guard expression h_k evaluating to `true`, the oce_k expression is evaluated. The transition

rule is

$$\frac{\begin{array}{c} \text{match}(u_1, \dots, u_m, d_1, \dots, d_m, \bullet, \rho') \\ \langle h_k, \varsigma, \rho' \rangle \Rightarrow \langle \text{true}, \epsilon, \varsigma, \rho'' \rangle \\ \langle \text{oce}_k, \varsigma, \rho'' \rangle \Rightarrow \langle \text{oe}', \epsilon, \varsigma, \rho''' \rangle \end{array}}{\langle \iota(d_1, \dots, d_m), \varsigma, \rho_0 \rangle \Rightarrow \langle \text{oe}', \epsilon, \varsigma, \rho_m \rangle}$$

where oe' must always be a tuple of form $\{\text{next_state}, \iota'(d'_1, \dots, d'_n)\}$. Similarly as for configuration access functions in **ERLANG/EFsm**, only the arguments are evaluated in an observer state, $\iota'(d'_1, \dots, d'_n)$.

3.6 The Observer Defined by an **ERLANG/OBS** Specification

Using the semantic definitions of the previous subsections, we can now define how an **ERLANG/OBS** specification defines an observer.

Definition 3.8 An **ERLANG/OBS** specification for an **ERLANG/EFsm** specification defines an observer, which is a tuple $\langle \Sigma, Q, q_0, Q_f, \rightarrow \rangle$, where

- Σ is the set of annotated computation steps of the **ERLANG/EFsm** specification,
- Q is the set of all terms of form $\iota(d_1, \dots, d_m)$, where ι is an observer location declared in the **ERLANG/OBS** specification, and d_1, \dots, d_m are Erlang values,
- q_0 is the term $\iota_0()$, where ι_0 is the declared initial observer location,
- Q_f is the set of all terms of form $\iota_f(d_1, \dots, d_m)$, where ι_f is a declared observer stop location, and
- $\rightarrow \subseteq (Q \times \Sigma \times Q)$ is the *observer step* relation. It consists of triples of form $\langle \iota(d_1, \dots, d_m), \varsigma, \iota'(d'_1, \dots, d'_{m'}) \rangle$, which we will write as

$$\iota(d_1, \dots, d_m) \xrightarrow{\varsigma} \iota'(d'_1, \dots, d'_{m'}),$$

such that the operational semantics allows to derive a transition of form

$$\langle \iota(d_1, \dots, d_m), \varsigma, \bullet \rangle \Rightarrow \langle \{\text{next_state}, \iota'(d'_1, \dots, d'_{m'})\}, \epsilon, \varsigma, \rho' \rangle.$$

□

We can now define what it means for a run or test case to cover a certain coverage item.

Definition 3.9 Let $\langle l_0, \bullet \rangle \xrightarrow{a_1(\overline{d_1})/\overline{b_1}} \dots \xrightarrow{a_n(\overline{d_n})/\overline{b_n}} \langle l_n, \sigma_n \rangle$ be a run of an **EFsm**, and for $i = 1, \dots, n$, let ς_i represent the annotated computation

step derived from the i th computation step, of form

$$\left\langle \begin{array}{l} \langle l_{i-1}, \sigma_{i-1} \rangle \xrightarrow{a_i(\overline{d_i})/\overline{b_i}} \langle l_i, \sigma_i \rangle, \\ l_{i-1}(a, u_1, \dots, u_m) \text{ when } g \rightarrow \overline{v} = \overline{e}, \overline{b}, \{\text{next_state}, l_i\} \end{array} \right\rangle$$

We say that the run *covers* a coverage item $q_f \in Q_f$ if there is a run $q_0 \xrightarrow{\varsigma_1} \dots \xrightarrow{\varsigma_n} q_n$ of the observer over $\varsigma_1 \dots \varsigma_n$ which ends in $q_f = q_n$. \square

Definition 3.10 A test case $\langle a_1(\overline{d_1})/\overline{b_1} \dots a_n(\overline{d_n})/\overline{b_n}, \Delta \rangle$ *covers* the coverage item q_f if the run induced by the test case covers q_f . \square

3.7 Symbolic semantics of ERLANG/OBS

In this section we give the operational semantics for ERLANG/OBS in the form of transition rules between *symbolic* structural states. Execution is symbolic in the sense that not all expressions in the EFSM can be evaluated to a value at the time of execution, since they contain symbolic parameters, whose values are not known at the time of evaluation. Thus, the observer may impose additional conditions, expressed in terms of symbolic parameters, on when a state can be reached. Recall, from Section 2.7, the definitions of symbolic structural state and symbolic transition rules for symbolic execution of an EFSM. The resulting expression of such symbolic evaluation is an ERLANG/EFSM normal form expression. Let \mathfrak{E} be the domain of ERLANG/EFSM normal form expressions. Let a *superposition condition* H be a ERLANG/EFSM guard expression, possibly containing symbolic parameters, but not containing local or global ERLANG/EFSM variables. A superposition condition is analogous to the path condition of Definition 2.5.

Definition 3.11 A *symbolic structural observer state* is a tuple

$$\langle oe, H, \epsilon, \varsigma^s, \rho \rangle$$

where

- oe is an ERLANG/OBS expression,
- H is a superposition condition, which is a condition on the symbolic parameters under which this symbolic structural state can be reached.
- ϵ is the empty sequence,
- $\varsigma^s \in V \rightarrow \mathfrak{E}$ is a global symbolic environment with bindings to the match variables V , and
- $\rho \in U \rightarrow \mathfrak{E}$ is a local symbolic environment with bindings to the observer variables U . \square

An initial symbolic structural observer state is a tuple $\langle \iota_0(), \text{true}, \epsilon, \varsigma^s, \bullet \rangle$ where $\iota_0()$ is an observer state and ς^s holds the bindings of match variables to expressions from the current symbolic annotated computation step.

Definition 3.12 A *symbolic annotated computation step* is a pair

$$\left\langle \begin{array}{l} \langle l, \sigma, G \rangle \xrightarrow{a(\bar{p})/\bar{b}} \langle l', \sigma', G' \rangle, \\ l(a, u_1, \dots, u_m) \text{ when } g \rightarrow \overline{v = \epsilon}, \bar{b}, \{\text{next_state}, l'\} \end{array} \right\rangle$$

that consists of

- a symbolic computation step $\langle l, \sigma, G \rangle \xrightarrow{a(\bar{p})/\bar{b}} \langle l', \sigma', G' \rangle$, and
- the (syntactic) **ERLANG/EFMS** edge clause

$$l(a, u_1, \dots, u_m) \text{ when } g \rightarrow \overline{v = \epsilon}, \bar{b}, \{\text{next_state}, l'\}$$

from which the symbolic computation step is derived. \square

A symbolic transition between two symbolic structural states

$$\langle oe, \varsigma^s, \rho \rangle \Rightarrow \langle \epsilon, H, \epsilon, \varsigma^s, \rho' \rangle$$

denotes that oe can be evaluated in a global environment ς^s and local environment ρ to an **ERLANG/EFMS** normal form expression ϵ , if the superposition condition H can be evaluated to **true**. For **ERLANG/OBS** with symbolic execution we reuse the match relation from Section 2.7.1.

Example 3.13 Define an observer predicate `equal_vars/2` to be true iff two different state variables `V1` and `V2` are bound to the same value before and after execution of the current symbolic annotated computation step. We use the match functions in Section 3.2.1 and define

```
equal_vars(V1,V2) ->
  lookup(V1,SourceEnv)==lookup(V2,SourceEnv) and
  lookup(V1,TargetEnv)==lookup(V2,TargetEnv) and
  (V1/=V2).
```

This means that the observer predicate evaluates to **true** if `V1` and `V2` are not the same state variable, and are bound to the same values in both match variables `SourceEnv` and `TargetEnv`. Assume an observer edge clause

$$\iota_0() \text{ when } \text{equal_vars}(V1, V2) \rightarrow oce.$$

Now, assume looking up values for `V1` and `V2` in `SourceEnv`

$$\varsigma^s(\text{SourceEnv})(V1) == \varsigma^s(\text{SourceEnv})(V2)$$

results in an expression $\mathbf{e}_1 == \mathbf{e}_2$, and similar, looking up values for $V1$ and $V2$ in TargetEnv

$$\varsigma^s(\text{TargetEnv})(V1) == \varsigma^s(\text{TargetEnv})(V2)$$

results in an expression $\mathbf{e}'_1 == \mathbf{e}'_2$ where both \mathbf{e}'_1 and \mathbf{e}'_2 are expressions with symbolic parameters. Then, given that $V1 \neq V2$ is satisfied, we can conclude that the necessary condition for the observer predicate to evaluate to **true** is that (the superposition condition) $\mathbf{e}_1 == \mathbf{e}_2 \wedge \mathbf{e}'_1 == \mathbf{e}'_2$ is satisfied. \square

3.7.1 Observer expressions

All expressions in **ERLANG/OBS** can be evaluated to **ERLANG/EFSM** normal form expressions with symbolic execution. In this section we outline the symbolic transition rules for how this is accomplished. For **ERLANG/EFSM** values, tuples, conses, logical operators, arithmetic expressions, observer variable definition and usage, and sequences of observer expressions *oce*, we use the symbolic transition rules from Section 2.7.2. Observer guard expressions compute boolean expressions of observer predicates and occur in observer edge clauses.

Similarly as in **ERLANG/EFSM** in Section 2.7.2, observer variables are also bound to values by pattern matching in observer expressions. Note that, in addition, observer variables are also bound to values by observer predicates, if not previously bound, see Section 3.5.2. Observer variables must be bound in the local environment, ρ , to an **ERLANG/EFSM** normal form expression before they are used.

3.7.2 Observer predicates

Assume an observer predicate is defined by a clause

$$f^{op}(u_1, \dots, u_n) \rightarrow ope.$$

f^{op} is the name of the observer predicate, u_1, \dots, u_n are observer variables and represent formal parameters to the observer predicate, and *ope* is a boolean expression, see Section 3.5.2. The symbolic transition rule is

$$\frac{\langle ope[u_1, \dots, u_n \mapsto obsge_1, \dots, obsge_n], \varsigma^s, \rho \rangle \Rightarrow \langle bool, H, \epsilon, \varsigma^s, \rho' \rangle}{\langle f^{op}(obsge_1, \dots, obsge_n), \varsigma^s, \rho \rangle \Rightarrow \langle bool, H, \epsilon, \varsigma^s, \rho' \rangle}$$

where the resulting local environment ρ' includes bindings to all unbound observer variables in $obsge_1, \dots, obsge_n$.

Pattern matching

The pattern match $cp=ope$ where cp is a pattern and ope an **ERLANG/EFSM** expression, binds any unbound observer variable in cp to a value, if pattern matching is possible. The match operator in **ERLANG/EFSM**, see Section 2.4.1, is only defined when a match succeeds between the two operands (and is not allowed in guards). However, in **ERLANG/OBS** we use the match operator in observer predicates (that occur in guards) and need to additionally consider the case when a match fails. A failing match implies a conflict between two expressions with bound variables. We therefore define an additional transition rule for pattern matching when all observer variables are bound to values and the pattern match fails, as follows

$$\frac{\langle cp==ope, \varsigma^s, \rho'' \rangle \Rightarrow \langle \text{false}, H, \epsilon, \varsigma^s, \rho' \rangle}{\langle cp=ope, \varsigma^s, \rho \rangle \Rightarrow \langle \text{false}, H, \epsilon, \varsigma^s, \rho' \rangle}$$

where the symbolic transition rule for the relational operator $==$ (from **ERLANG/EFSM**) is only defined when all its operands are bound to values. Thus, a failing pattern match only has a symbolic transition when there is no unbound variables in cp .

Match functions

Match functions are used to help define observer predicates. With symbolic execution match functions may require additional superposition conditions to be satisfied, in order to fully evaluate all expressions. We will assume that in general match functions can be defined freely and allow for definitions with any valid **ERLANG** expression. We do this with the assumption that the match functions, during symbolic execution, are executed analogously as user defined functions of **ERLANG/EFSM** (see Section 2.7), i.e., such that

- each argument in a call to the match function generate a superposition condition,
- for each evaluated **ERLANG** clause, a superposition condition is generated for each pattern match in the function head, for the execution of the guard, and for the execution of the body of the **ERLANG** clause.

The resulting conjunction of these superposition conditions is the contributed superposition condition of the match function.

In Figure 3.8 we give an assumed representation of match variables, with symbolic execution of the **EFSM**, in terms of **ERLANG/EFSM** expressions. Similarly as in Section 3.5.2, this is the representation assumed in the definitions of the match functions. We distinguish between binding and non-binding match functions. For match functions that bind any unbound observer variable and appear in the thesis, all the necessary transition rules are given below. For match functions that assume all

observer variables used are already bound, transitions rules are straightforward and we omit them here.

Match variables for computation steps: $\langle l, \sigma, G \rangle \xrightarrow{a(\overline{p})/\overline{b}} \langle l', \sigma', G' \rangle$	
Eventtype	<i>atom</i>
Eventpars	$[p_1, \dots, p_n]$
Outevents	$[\#func\{op=b_1, args=[e_{11}, \dots, e_{1m}]\}$
	\vdots
	$\#func\{op=b_n, args=[e_{n1}, \dots, e_{nm}]\}$]
SourceLoc	<i>atom</i> ,
SourceEnv	$[\{v_1, e_1\} \cdots \{v_n, e_n\}]$
TargetLoc	<i>atom</i> ,
TargetEnv	$[\{v_1, e_1\} \cdots \{v_n, e_n\}]$
Match variables for ERLANG/EFMSM edge clauses:	
$l(a, u_1, \dots, u_m) \text{ when } g \rightarrow \overline{v} = \overline{e}, \overline{b}, \{\text{next_state}, l'\}$	
Edgevars	$[u_1, \dots, u_m]$
Guard	$\#func\{op=guardop, args=[ge_1, \dots, ge_n]\}$
Assigns	$[\{v_1, e_1\} \cdots \{v_n, e_n\}]$
Outexprs	$[\#func\{op=b_1, args=[e_{11}, \dots, e_{1m}]\}$
	\vdots
	$\#func\{op=b_n, args=[e_{n1}, \dots, e_{nm}]\}$]

Figure 3.8. Representation of the match variables in Figure 3.4 as ERLANG/EFMSM expressions with symbolic execution.

Binding match functions

In this section we describe match functions that bind unbound observer variables to normal form expressions. It can here be noted that with symbolic execution, match functions can only bind observer variables to values and ERLANG/EFMSM expressions.

`member(Element, List)`

A predicate such that `member(Element, List)` returns true if *Element* is a member of the list *List* (with length $n \geq 0$) and false otherwise. The

symbolic transition rules for **member** are as follows.

$$\frac{\begin{array}{l} \langle e_1, \varsigma^s, \rho \rangle \Rightarrow \langle \mathbf{e}, H', \epsilon, \varsigma^s, \rho \rangle \\ \langle e_2, \varsigma^s, \rho \rangle \Rightarrow \langle [\mathbf{e}_1, \dots, \mathbf{e}_n], H'', \epsilon, \varsigma^s, \rho \rangle \\ \exists i : 1 \leq i \leq n :: \text{match}(\mathbf{e}, \mathbf{e}_i, \rho, H_i, \rho) \end{array}}{\langle \text{member}(e_1, e_2), \varsigma^s, \rho \rangle \Rightarrow \langle \text{true}, H' \wedge H'' \wedge H_i, \epsilon, \varsigma^s, \rho \rangle}$$

if e_1 is an expression that evaluates to a value or a bound observer variable, which is a member of a list e_2 ,

$$\frac{\begin{array}{l} \langle e_1, \varsigma^s, \rho \rangle \Rightarrow \langle \mathbf{e}, H', \epsilon, \varsigma^s, \rho \rangle \\ \langle e_2, \varsigma^s, \rho \rangle \Rightarrow \langle [\mathbf{e}_1, \dots, \mathbf{e}_n], H'', \epsilon, \varsigma^s, \rho \rangle \\ \forall i, 1 \leq i \leq n : \neg \text{match}(\mathbf{e}, \mathbf{e}_i, \rho, H_i, \rho) \end{array}}{\langle \text{member}(e_1, e_2), \varsigma^s, \rho \rangle \Rightarrow \langle \text{false}, H' \wedge H'' \wedge H_1 \wedge \dots \wedge H_n, \epsilon, \varsigma^s, \rho \rangle}$$

if e_1 is an expression that evaluates to a value or a bound observer variable, which is *not* a member of a list e_2 ,

$$\frac{\begin{array}{l} \langle e_1, \varsigma^s, \rho \rangle \Rightarrow \langle u, H, \epsilon, \varsigma^s, \rho \rangle \\ u \text{ is an observer variable, not defined in } \rho \\ \langle e_2, \varsigma^s, \rho \rangle \Rightarrow \langle [], \text{true}, \epsilon, \varsigma^s, \rho \rangle \end{array}}{\langle \text{member}(e_1, e_2), \varsigma^s, \rho \rangle \Rightarrow \langle \text{false}, H, \epsilon, \varsigma^s, \rho[u \mapsto \text{undefined}] \rangle}$$

if e_1 is an expression that evaluates to an unbound observer variable, e_2 is the empty list, and

$$\frac{\begin{array}{l} \langle e_1, \varsigma^s, \rho \rangle \Rightarrow \langle u, H, \epsilon, \varsigma^s, \rho \rangle \\ u \text{ is an observer variable, not defined in } \rho \\ \langle e_2, \varsigma^s, \rho \rangle \Rightarrow \langle [\mathbf{e}_1, \dots, \mathbf{e}_n], \text{true}, \epsilon, \varsigma^s, \rho \rangle \end{array}}{\langle \text{member}(e_1, e_2), \varsigma^s, \rho \rangle \Rightarrow \langle \text{true}, H, \epsilon, \varsigma^s, \rho[u \mapsto \mathbf{e}_i] \rangle}$$

if e_1 is an expression that evaluates to an unbound observer variable and e_2 is a non-empty list. In this case *any* member of e_2 is a possible solution. Note that we do not define any symbolic transition rule for the case when e_2 is an unbound observer variable as that would generate an infinite number of possible solutions.

lookup(*Key*, *KeyValList*)

Searches a list of tuples, *KeyValList* (with length $n \geq 0$), for a tuple whose 1:st element matches **Key**. If found, the 2:nd element of such a tuple is returned, otherwise the boolean **false** is returned. The symbolic

transition rules for `lookup` are as follows.

$$\begin{array}{c}
\langle e_1, \varsigma^s, \rho \rangle \Rightarrow \langle \mathbf{e}_1, H', \epsilon, \varsigma^s, \rho \rangle \\
\langle e_2, \varsigma^s, \rho \rangle \Rightarrow \langle [\{\mathbf{e}_{11}, \mathbf{e}_{12}\}, \dots, \{\mathbf{e}_{n1}, \mathbf{e}_{n2}\}], H'', \epsilon, \varsigma^s, \rho \rangle \\
\forall j, 1 \leq j < i : \neg \text{match}(\mathbf{e}_1, \mathbf{e}_{j1}, \rho, H_{j1}, \rho) \\
\text{match}(\mathbf{e}_1, \mathbf{e}_{i1}, \rho, H_{i1}, \rho) \\
\hline
\langle \text{lookup}(e_1, e_2), \varsigma^s, \rho \rangle \Rightarrow \langle \mathbf{e}_{i2}, H' \wedge H'' \wedge H_{11} \wedge \dots \wedge H_{i1}, \epsilon, \varsigma^s, \rho \rangle
\end{array}$$

if e_1 is a value or a bound observer variable, which is a key in the key/value list e_2 ,

$$\begin{array}{c}
\langle e_1, \varsigma^s, \rho \rangle \Rightarrow \langle \mathbf{e}_1, H', \epsilon, \varsigma^s, \rho \rangle \\
\langle e_2, \varsigma^s, \rho \rangle \Rightarrow \langle [\{\mathbf{e}_{11}, \mathbf{e}_{12}\}, \dots, \{\mathbf{e}_{n1}, \mathbf{e}_{n2}\}], H'', \epsilon, \varsigma^s, \rho \rangle \\
\forall j, 1 \leq j \leq n : \neg \text{match}(\mathbf{e}_1, \mathbf{e}_{j1}, \rho, H_{ji}, \rho) \\
\hline
\langle \text{lookup}(e_1, e_2), \varsigma^s, \rho \rangle \Rightarrow \langle \text{false}, H' \wedge H'' \wedge H_{11} \wedge \dots \wedge H_{n1}, \epsilon, \varsigma^s, \rho \rangle
\end{array}$$

if e_1 is a value or a bound observer variable, which is *not* a key in the key/value list e_2 ($n \geq 0$), and

$$\begin{array}{c}
\langle e_1, \varsigma^s, \rho \rangle \Rightarrow \langle u, H', \epsilon, \varsigma^s, \rho \rangle \\
u \text{ is an observer variable, not defined in } \rho \\
\langle e_2, \varsigma^s, \rho \rangle \Rightarrow \langle [\{\mathbf{e}_{11}, \mathbf{e}_{12}\}, \dots, \{\mathbf{e}_{n1}, \mathbf{e}_{n2}\}], \text{true}, \epsilon, \varsigma^s, \rho \rangle \\
\hline
\langle \text{lookup}(e_1, e_2), \varsigma^s, \rho \rangle \Rightarrow \langle \mathbf{e}_{i2}, \text{true}, \epsilon, \varsigma^s, \rho[u \mapsto \mathbf{e}_{i2}] \rangle
\end{array}$$

if e_1 is an expression that evaluates to an unbound observer variable and e_2 is a non-empty key/value list. In this case *any* member of e_2 is a possible solution. If e_2 evaluates to an empty list ($n = 0$), u is bound to **undefined** and **false** returned. Note that we do not define any symbolic transition rule for the case when e_2 is an unbound observer variable as that would generate an infinite number of possible solutions.

3.7.3 Observer edge clauses

The symbolic transitions from an observer state $\iota(\mathbf{e}_1, \dots, \mathbf{e}_m)$ are defined by observer edge clauses

$$\begin{array}{c}
\iota(u_1, \dots, u_m) \text{ when } h_1 \rightarrow o c e_1; \\
\vdots \\
\iota(u_1, \dots, u_m) \text{ when } h_n \rightarrow o c e_n.
\end{array}$$

Each of these clauses are evaluated by first matching the expressions $\mathbf{e}_1, \dots, \mathbf{e}_m$ with the observer variables u_1, \dots, u_m , and if the match succeeds the observer guard expression h_k is evaluated. For *some* clause

k with all observer variables, u_1, \dots, u_m matching and observer guard expression h_k evaluating to **true**, the oce_k expression is evaluated. The symbolic transition rule is

$$\frac{\text{match}(u_{k1}, \dots, u_{km}, \mathbf{e}_1, \dots, \mathbf{e}_m, \bullet, \rho') \quad \langle h_k, \varsigma^s, \rho \rangle \Rightarrow \langle \text{true}, H, \epsilon, \varsigma^s, \rho' \rangle \quad \langle oce_k, \varsigma^s, \rho'' \rangle \Rightarrow \langle \mathbf{e}', \text{true}, \epsilon, \varsigma^s, \rho''' \rangle}{\langle \iota(\mathbf{e}_1, \dots, \mathbf{e}_m), \varsigma^s, \rho_0 \rangle \Rightarrow \langle \mathbf{e}', H, \epsilon, \varsigma^s, \rho_m \rangle}$$

where \mathbf{e}' always must be a tuple of form $\{\text{next_state}, \iota'(\mathbf{e}'_1, \dots, \mathbf{e}'_n)\}$.

3.7.4 Correspondence with Operational Semantics

In analogy with the treatment for EFSMs in Section 2.8, we can derive a similar relationship between (non-symbolic) operational semantics and symbolic semantics for observers. First, we note that structural observer states and transitions in the (non-symbolic) operational semantics is defined for global environments ς that map match variables to components of computation steps. In contrast, a global environment ς^s in the symbolic semantics maps match variables to corresponding components of *symbolic* computation steps. We observe that the domains of ς and ς^s are the same. Moreover, if ς^s represents a symbolic version of a computation step represented by ς , then ς and ς^s are related by the correspondence relation \triangleleft_Γ . More precisely, if ς represents a computation step

$$\langle l, \sigma \rangle \xrightarrow{a(\bar{d})/\bar{b}} \langle l', \sigma' \rangle$$

and ς^s represents a symbolic computation step

$$\langle l, \sigma, G \rangle \xrightarrow{a(\bar{p})/\bar{b}} \langle l', \sigma', G' \rangle,$$

such that $\langle l, \sigma \rangle \triangleleft_\Gamma \langle l, \sigma_s, G \rangle$, $\bar{d} = \Gamma(\bar{p})$, $\bar{b} \triangleleft_\Gamma \bar{b}$, and $\langle l', \sigma' \rangle \triangleleft_\Gamma \langle l', \sigma'_s, G' \rangle$, then $\varsigma \triangleleft_\Gamma \varsigma^s$. This correspondence can be checked by noting the close correspondence between the definitions of match variables in the non-symbolic case (Fig. 3.4) and symbolic case (Fig. 3.8).

The definition of the correspondence relation \triangleleft_Γ between structural states and symbolic structural states directly applies also to observer structural states. That is, $\langle oe, \epsilon, \varsigma, \rho \rangle \triangleleft_\Gamma \langle oe_s, H, \epsilon, \varsigma^s, \rho_s \rangle$ is defined precisely as in Section 2.8.

We can now derive an analogue of Proposition 2.7.

Proposition 3.14 Assume that $\langle oe, \epsilon, \varsigma, \rho \rangle \triangleleft_\Gamma \langle oe_s, \text{true}, \epsilon, \varsigma^s, \rho_s \rangle$. Then

- (i) for each observer transition $\langle oe, \varsigma, \rho \rangle \Rightarrow \langle d, \epsilon, \varsigma, \rho' \rangle$ there is a symbolic observer transition $\langle oe_s, \varsigma^s, \rho_s \rangle \Rightarrow \langle \mathbf{e}, H, \epsilon, \varsigma^s, \rho'_s \rangle$ such that $\langle d, \epsilon, \varsigma, \rho' \rangle \triangleleft_\Gamma \langle \mathbf{e}, H, \epsilon, \varsigma^s, \rho'_s \rangle$,
- (ii) for each symbolic observer transition $\langle oe_s, \varsigma^s, \rho_s \rangle \Rightarrow \langle \mathbf{e}, H, \epsilon, \varsigma^s, \rho'_s \rangle$ such that $\Gamma \vdash H$, there is an observer transition $\langle oe, \varsigma, \rho \rangle \Rightarrow \langle d, \epsilon, \varsigma, \rho' \rangle$ such that $\langle d, \epsilon, \varsigma, \rho' \rangle \triangleleft_\Gamma \langle \mathbf{e}, H, \epsilon, \varsigma^s, \rho'_s \rangle$. \square

As in Proposition 2.7, Property (i) states that each transition in the non-symbolic semantics has a corresponding transition in the symbolic semantics, and Property (ii) states that each transition in the symbolic semantics, which is consistent with the parameter environment in the sense that H is satisfied, has a corresponding transition in the non-symbolic semantics.

3.8 Symbolic Observers and Symbolic Coverage

In this section, we will provide a symbolic version of the observer definition in Section 3.6. The section is thus analogous with Section 2.9. We assume that the observer is defined for an EFSM, defined by an **ERLANG/EFMS** definition, with symbolic semantics as in Section 2.9.

Let a *symbolic observer state* \mathbf{q} be a term of the form $\iota(\mathbf{e}_1, \dots, \mathbf{e}_n)$, where ι is an observer location declared in the **ERLANG/OBS** specification and $\mathbf{e}_1, \dots, \mathbf{e}_n$ are **ERLANG/EFMS** normal form expressions. We say that the symbolic observer state is *initial (accepting)* if ι is initial (accepting). Let a *conditioned observer state* be a term of form $\langle \mathbf{q}, H \rangle$, where \mathbf{q} is a symbolic observer state and H is a superposition condition. We say that the conditioned observer state $\langle \mathbf{q}, H \rangle$ is *accepting* if \mathbf{q} is accepting. The initial conditioned observer state is $\langle \iota_0(), \text{true} \rangle$.

The symbolic version of the observer step relation is represented by triples of form

$$\langle \iota(\mathbf{e}_1, \dots, \mathbf{e}_m), H \rangle \xrightarrow{\varsigma^s} \langle \iota'(\mathbf{e}'_1, \dots, \mathbf{e}'_n), H \wedge H' \rangle,$$

such that the rules in the symbolic semantics for observers allow to derive a symbolic transition

$$\langle \iota(\mathbf{e}_1, \dots, \mathbf{e}_m), \varsigma^s, \bullet \rangle \Rightarrow \langle \{\text{next_state}, \iota'(\mathbf{e}'_1, \dots, \mathbf{e}'_n)\}, H', \epsilon, \varsigma^s, \rho \rangle$$

for some ρ . Intuitively, whenever the observer is in a conditioned observer state $\langle \iota(\mathbf{e}_1, \dots, \mathbf{e}_m), H \rangle$ it can process a symbolic annotated computation step, represented by the environment ς^s , and move to a next conditioned observer state $\langle \iota'(\mathbf{e}'_1, \dots, \mathbf{e}'_n), H \wedge H' \rangle$.

We can then define a correspondence with the (non-symbolic) observer step relation, and derive an analogue of Proposition 2.10. Define $q \triangleleft_\Gamma \langle \mathbf{q}, H \rangle$ to denote that $\Gamma \vdash H$ and $q \triangleleft_\Gamma \mathbf{q}$.

Proposition 3.15 Assume $q \triangleleft_\Gamma \langle \mathbf{q}, H \rangle$ for a parameter environment Γ . Then

- (i) whenever $q \xrightarrow{\varsigma} q'$ and $\varsigma \triangleleft_\Gamma \varsigma^s$, there is a symbolic observer step $\langle \mathbf{q}, H \rangle \xrightarrow{\varsigma^s} \langle \mathbf{q}', H' \rangle$ with $q' \triangleleft_\Gamma \langle \mathbf{q}', H' \rangle$,
- (ii) whenever $\langle \mathbf{q}, H \rangle \xrightarrow{\varsigma^s} \langle \mathbf{q}', H' \rangle$ such that $\Gamma \vdash H'$, then there is an observer step $q \xrightarrow{\varsigma} q'$ with $\varsigma \triangleleft_\Gamma \varsigma^s$, and $q' \triangleleft_\Gamma \langle \mathbf{q}', H' \rangle$. \square

Proposition 3.15 follows rather straight-forwardly from Proposition 3.14 and the definitions of non-symbolic and symbolic observer steps.

We can also provide a definition of coverage for the symbolic case.

Definition 3.16 Let $\langle l_0, \bullet, \text{true} \rangle \xrightarrow{a_1(\overline{p_1})/\overline{b_1}} \dots \xrightarrow{a_n(\overline{p_n})/\overline{b_n}} \langle l_n, \sigma_n, G_n \rangle$ be a symbolic run of an EFSM, and for $i = 1, \dots, n$, let ς_i^s represent the symbolic annotated computation step

$$\left\langle \begin{array}{l} \langle l_{i-1}, \sigma_{i-1}, G_{i-1} \rangle \xrightarrow{a_i(\overline{p_i})/\overline{b_i}} \langle l_i, \sigma_i, G_i \rangle, \\ l_{i-1}(a, u_1, \dots, u_m) \text{ when } g \rightarrow \overline{v} = \overline{e}, \overline{b}, \{\text{next_state}, l_i\} \end{array} \right\rangle$$

derived from the i step in this symbolic run. Let \mathbf{q}_f be an accepting symbolic observer state. Then we say that this symbolic run *symbolically covers* $\langle \mathbf{q}_f, H_n \rangle$ if there is a sequence of symbolic observer steps $\langle \mathbf{q}_0, \text{true} \rangle \xrightarrow{\varsigma_1^s} \dots \xrightarrow{\varsigma_n^s} \langle \mathbf{q}_n, H_n \rangle$ (i.e., a *symbolic run* over $\varsigma_1^s \dots \varsigma_n^s$) such that $\mathbf{q}_n = \mathbf{q}_f$. \square

Each covered conditioned observer state, of the form $\langle \iota(\overline{e}), H_n \rangle$, represents the set of coverage items $\iota(\overline{d})$ that can be obtained by finding a parameter environment so that \overline{e} becomes \overline{d} , and so that G_n and H_n are satisfied. Thus, for each feasible tuple of values \overline{d} , a test case that covers $\iota(\overline{d})$ can be obtained from the symbolic trace w by finding a parameter environment Γ so that $\Gamma \vdash (G_n \wedge H_n \wedge \overline{e} = \overline{d})$. We summarize this by saying that the symbolic test case $\langle w, (G_n \wedge H_n \wedge \overline{e} = \overline{d}) \rangle$ *covers* $\iota(\overline{d})$.

Intuitively, these definitions says that all coverage items represented by an instantiation of \mathbf{q}_f that satisfies $G_n \wedge H_n$, are covered by runs of the EFSM that instantiate the symbolic run $\langle l_0, \bullet, \text{true} \rangle \xrightarrow{a_1(\overline{p_1})/\overline{b_1}} \dots \xrightarrow{a_n(\overline{p_n})/\overline{b_n}} \langle l_n, \sigma_n, G_n \rangle$ in a way that satisfies $G_n \wedge H_n$.

Th correspondence between coverage of (non-symbolic) runs and symbolic runs is made precise by the following theorem.

Theorem 3.17 Assume an EFSM and an observer monitoring the EFSM. Then there is a run of the EFSM which covers a coverage item $\iota_f(\overline{d})$ if and only if there is a symbolic run of the EFSM with final path condition G_n , which symbolically covers an accepting conditioned observer state $\langle \iota_f(\overline{e}), H_n \rangle$ for some parameter environment Γ such that $\Gamma \vdash (G_n \wedge H_n \wedge \overline{e} = \overline{d})$. \square

Proof of Theorem 3.17: First assume that there is a run

$$\Delta \models \langle l_0, \bullet \rangle \xrightarrow{a_1(\overline{d_1})/\overline{b_1}} \dots \xrightarrow{a_n(\overline{d_n})/\overline{b_n}} \langle l_n, \sigma_n \rangle$$

of the EFSM which covers q_f . By repeated applications of case (i) of Proposition 2.10, we infer that there is a corresponding symbolic run

$$\langle l_0, \bullet, \text{true} \rangle \xrightarrow{a_1(\overline{p_1})/\overline{b_1}} \dots \xrightarrow{a_n(\overline{p_n})/\overline{b_n}} \langle l_n, \sigma_n^s, G_n \rangle$$

and a parameter environment Γ with $\Delta = \Delta_\Gamma$, such that $\langle l_i, \sigma_i \rangle \triangleleft_\Gamma \langle l_i, \sigma_i^s, G_i \rangle$ for $i = 0, \dots, n$, and $\overline{p_i} \triangleleft_\Gamma \overline{d_i}$ and $\overline{b_i} \triangleleft_\Gamma \overline{b_i}$ for $i = 1, \dots, n$. Note that $\langle l_i, \sigma_i \rangle \triangleleft_\Gamma \langle l_i, \sigma_i^s, G_i \rangle$ implies that Γ satisfies G_i . for $i = 0, \dots, n$.

For $i = 1, \dots, n$, let ς_i represent the annotated computation step corresponding to the i th step in the above run, and let ς_i^s represent the symbolic annotated computation step corresponding to the i th step in the above symbolic run. The correspondences between the two runs means that $\varsigma_i \triangleleft_\Gamma \varsigma_i^s$.

By the definition of coverage, there is a run $q_0 \xrightarrow{\varsigma_1} \dots \xrightarrow{\varsigma_n} q_f$ of the observer over $\varsigma_1 \dots \varsigma_n$, which ends in q_f . By repeated application of case (i) of Proposition 3.15, there is then a symbolic run $\langle q_0, \text{true} \rangle \xrightarrow{\varsigma_1^s} \dots \xrightarrow{\varsigma_n^s} \langle q_n, H_n \rangle$ of the observer over $\varsigma_1^s \dots \varsigma_n^s$, such that $q_i \triangleleft_\Gamma \langle q_i, H_i \rangle$ for $i = 0, \dots, n$. Let q_f be $\iota_f(\overline{d})$ and let q_f be $\iota_f(\overline{e})$. This implies that the above symbolic run of the EFSM symbolically covers $\langle q_f, H_n \rangle$, and moreover by used definitions we have $\Gamma \vdash (G_n \wedge H_n \wedge \overline{e} = \overline{d})$.

In the other direction, assume that there is a symbolic run of the EFSM

$$\langle l_0, \bullet, \text{true} \rangle \xrightarrow{a_1(\overline{p_1})/\overline{b_1}} \dots \xrightarrow{a_n(\overline{p_n})/\overline{b_n}} \langle l_n, \sigma_n, G_n \rangle$$

which symbolically covers $\langle q_f, H_n \rangle$, such that $\Gamma \vdash (G_n \wedge H_n \wedge \overline{e} = \overline{d})$, where q_f is $\iota_f(\overline{e})$. Let q_f be $\iota_f(\overline{d})$. This means that there is a symbolic run $\langle q_0, \text{true} \rangle \xrightarrow{\varsigma_1^s} \dots \xrightarrow{\varsigma_n^s} \langle q_n, H_n \rangle$ of the observer over $\varsigma_1^s \dots \varsigma_n^s$, where ς_i^s is the symbolic annotated computation that represents the i th step in the symbolic run of the EFSM. For $i = 1, \dots, n$, let $\overline{d_i} = \Gamma(\overline{p_i})$ and $\overline{b_i} \triangleleft_\Gamma \overline{b_i}$. By repeated application of case (ii) of Proposition 2.10, we infer that there is a run of the EFSM

$$\Delta \models \langle l_0, \bullet \rangle \xrightarrow{a_1(\overline{d_1})/\overline{b_1}} \dots \xrightarrow{a_n(\overline{d_n})/\overline{b_n}} \langle l_n, \sigma_n \rangle$$

such that $\langle l_i, \sigma_i \rangle \triangleleft_\Gamma \langle l_i, \sigma_i^s, G_i \rangle$ for $i = 0, \dots, n$. Note that $\langle l_i, \sigma_i \rangle \triangleleft_\Gamma \langle l_i, \sigma_i^s, G_i \rangle$ implies that Γ satisfies G_i . for $i = 0, \dots, n$. The correspondence between the two runs also means that $\varsigma_i \triangleleft_\Gamma \varsigma_i^s$ for $i = 1, \dots, n$. Thus, by repeated application of case (ii) of Proposition 3.15, we infer that

there is a run $q_0 \xrightarrow{\varsigma_1} \dots \xrightarrow{\varsigma_n} q_f$ of the observer over $\varsigma_1 \dots \varsigma_n$, which ends in $\iota_f(\bar{d})$. By the definition of coverage, this means that the above run of the EFSM indeed covers $\iota_f(\bar{d})$, and the theorem is proven. \square

The following example illustrates Theorem 3.17.

Example 3.18 Consider the EFSM defined in Example 2.1 and the “all target locations of the EFSM” observer from Example 3.1. A possible symbolic run of the EFSM is

$$\begin{aligned}
& \langle \text{morning}, [], \text{true} \rangle \xrightarrow{\text{wakeup}(TDay)/\text{checkout}(TDay)} \\
& \langle \text{workUU}, \left[\begin{array}{l} \text{Progress}=0 \\ \text{Stamina}=2 \\ \text{Day}=TDay \end{array} \right], \text{daytype}(TDay)=\text{collect} \rangle \xrightarrow{\text{incident}(I_1)/\text{checkin}(TDay)} \\
& \langle \text{preschool}, \left[\begin{array}{l} \text{Progress}=0 \\ \text{Stamina}=1 \\ \text{Day}=TDay \end{array} \right], \text{daytype}(TDay)=\text{collect} \wedge I_1 = \text{kids_happy} \rangle \xrightarrow{\text{incident}(I_2)/} \\
& \langle \text{end_of_day}, \left[\begin{array}{l} \text{Progress}=0 \\ \text{Stamina}=0 \\ \text{Day}=TDay \end{array} \right], \text{daytype}(TDay)=\text{collect} \wedge I_1=\text{kids_happy} \wedge I_2=\text{kids_notok} \rangle
\end{aligned}$$

There is a symbolic run of the observer in Example 3.1 over this EFSM-run which symbolically covers the three conditioned observer states

$$\begin{aligned}
& \langle \text{visitedloc}(\text{workUU}), \text{true} \rangle, \\
& \langle \text{visitedloc}(\text{preschool}), \text{true} \rangle \text{ and} \\
& \langle \text{visitedloc}(\text{end_of_day}), \text{true} \rangle
\end{aligned}$$

Thus, these conditioned observer states are covered by the symbolic test case $\langle w, G \rangle$, where

- w is the symbolic trace with input and output expressions:

$$\begin{aligned}
& \text{wakeup}(TDay)/\text{checkout}(TDay) \\
& \text{incident}(I_1)/\text{checkin}(TDay) \\
& \text{incident}(I_2)/,
\end{aligned}$$

- G is the path condition:

$$\text{daytype}(TDay)=\text{collect} \wedge I_1 = \text{kids_happy} \wedge I_2 = \text{kids_notok}.$$

An abstract test case, which satisfies the path condition can be trivially extracted. \square

Example 3.19 Assume an edge clause

$$l(a) \text{ when } (X==0) \text{ or } (Y==2) \rightarrow \{\text{next_state}, l'\}$$

where X and Y are state variables that both assume values in $\{0, 1, 2, 3\}$. From Section 4.1.1 we have that Condition coverage requires each condition in a guard to be evaluated to true or false. An observer implementing this requirement has four possible symbolic transitions with observer guards:

$$\begin{aligned} h_1 &= \text{is_cc}('X==0', \text{false}) \\ h_2 &= \text{is_cc}('X==0', \text{true}) \\ h_3 &= \text{is_cc}('Y==2', \text{false}) \\ h_4 &= \text{is_cc}('Y==2', \text{true}) \end{aligned}$$

In a symbolic execution, let ς^s be the current bindings of the match variables and assume the state variable X is evaluated to the symbolic parameter p_1 and the state variable Y is evaluated to the symbolic parameter p_2 . Then we have the corresponding superposition condition results

$$\begin{aligned} H_1 &= p_1 \neq 0 \\ H_2 &= p_1 = 0 \\ H_3 &= p_2 \neq 2 \\ H_4 &= p_2 = 2 \end{aligned}$$

It follows that only pairs of observer guards with consistent H_i can be performed simultaneously. Assuming a bit vector organized as $\langle h_1, h_2, h_3, h_4 \rangle$, the set of possible binary vectors is then $\{1010, 1001, 0110, 0101\}$. Thus for e.g., 0101, we have ' $X==0$ ', ' $Y==2$ ', and

$$\bigwedge_{i=1,3} \neg(\varsigma^s \vdash H_i) \wedge \bigwedge_{i=2,4} \varsigma^s \vdash H_i \text{ and } 1 \leq i \leq 4$$

evaluating to true. I.e., ς^s satisfy superposition conditions H_2 and H_4 , but does not satisfy superposition conditions H_1 and H_3 . \square

4. Coverage criteria

Observer automata, introduced in the previous chapter, control the selection of test cases. In this chapter we give examples of how observer automata can be used to define a variety of coverage criteria.

We structure coverage criteria into two classes.

- *Model-independent coverage criteria*, which require only limited knowledge about the actual model and thus can easily be reused between different, unrelated models, are discussed in Section 4.1.
- *Model-dependent coverage criteria*, which depend on a specific model and thus, in general, cannot be reused on other models, are discussed in Section 4.2.

4.1 Model-independent coverage criteria

This section describes how a number of well-known coverage criteria can be specified with observers. In the literature [BJK 04, Utting 07, Ammann 08] coverage criteria are often divided into control flow and data flow criteria. Control flow criteria can be further divided into coverage of guards (Section 4.1.1), locations (Section 4.1.2) and sequences of edges (Section 4.1.3). Examples of data flow criteria are given in Section 4.1.4.

4.1.1 Coverage of guards

Several coverage criteria are based on logical expressions that determine whether a certain path of execution should be taken. In the literature, a *decision* is often referred to as a boolean expression that control program flow, and each predicate in such expressions as a *condition*. For example consider the following C/Java/JavaScript/... code fragment:

```
if (decision)
    statement1;
else
    statement2;
statement3;
```

In this thesis we evaluate transition clauses into edge clauses and thereby possibly evaluate several decisions and conditions already before a run of

the EFSM. We therefore use the term decision as a synonym for the guard of an edge, i.e., the resulting expressions after evaluating all decisions in the transition clause.

To illustrate these criteria we assume an edge clause (see Section 2.10)

$$l(a) \text{ when } (X==0) \text{ or } (Y==2) \rightarrow \{\text{next_state}, l'\}$$

where the guard, $(X==0) \text{ or } (Y==2)$, is a decision, and $X==0$ and $Y==2$ are two conditions. Further, we assume that both X and Y are state variables that assume values in $\{0, 1, 2, 3\}$.

Below we outline some classic coverage criteria, originally used for white-box testing.

X	Y	X	Y	X	Y	X	Y	X	Y	X	Y
0	2	–	–	0	2	–	–	–	–	–	–
–	–	0	3	0	3	0	3	0	3	0	3
–	–	1	2	1	2	1	2	1	2	1	2
1	3	–	–	1	3	1	3	1	3	1	3
(i)		(ii)		(iii)		(iv)		(v)			

Figure 4.1. Sufficient tests for a decision $(X==0) \text{ or } (Y==2)$ with (i) Decision Coverage, (ii) Condition Coverage, (iii) Multiple Condition Coverage, (iv) Condition/Decision Coverage and (v) Modified Condition/Decision Coverage.

Decision Coverage (DC) [Myers 79] is also known as *Branch coverage* and requires two test cases for a decision d . One test case when d evaluates to false and one when d evaluates to true. For example, considering the decision $(X==0) \text{ or } (Y==2)$ in the edge clause above, the two tests in Figure 4.1(i) are sufficient.

Condition Coverage (CC) [Myers 79] requires two test cases for each condition c in a decision: one test case in which c evaluates to false, and one in which c evaluates to true. For example, considering the decision $(X==0) \text{ or } (Y==2)$ in the edge clause above, the two tests in Figure 4.1(ii) are sufficient. For a decision with n conditions, at most $2n$ test cases are required.

Multiple Condition Coverage (MCC) [Myers 79] is also known as *Combinatorial Coverage* and requires one test case for each possible combination of evaluations of the conditions in the decision. Thus, for a decision with n boolean conditions, at most 2^n test cases are required. For example, considering the decision $(X==0) \text{ or } (Y==2)$ in the edge clause above, the four tests in Figure 4.1(iii) are sufficient.

Condition Coverage/Decision Coverage (CC/DC) [Myers 79] requires both Condition Coverage and Decision Coverage to be satisfied. For ex-

ample, considering the decision $(X==0)$ or $(Y==2)$ in the edge clause above, the three tests in Figure 4.1(iv) are sufficient.

Modified Condition/Decision Coverage (MC/DC) [Chilenski 94] requires the outcome of a decision to *depend* on a single condition while all other conditions in the decision are fixed. Modified Condition/Decision Coverage requires two test cases for each condition c in a decision d that independently affect the decision's outcome. One test case when c evaluates to false and one when c evaluates to true. Among the constructed test cases, at least one test case must evaluate d to false and one must evaluate d to true. For example, considering the decision $(X==0)$ or $(Y==2)$ in the edge clause above, the three tests in Figure 4.1(v) are sufficient. In general, for a decision with n conditions, the number of required test cases ranges between $n + 1$ and $2n$.

Due to dependencies between conditions it may not be possible to change the outcome of one condition while keeping all other conditions. Several variants have therefore been developed relaxing this requirement, see e.g., [Ntafos 88, Ammann 03] for further examples of coverage criteria on logical expressions.

We will apply the above coverage criteria on **ERLANG/EFM**S. In Section 2.10 we defined edge clauses in terms of symbolically executed transition clauses, and in Section 5 we will see that a test case corresponds to a sequence of edge clauses. Further, for any two edge clauses with the same source location reacting on the same input event the corresponding guards must be inconsistent, for the EFSM to be deterministic. This influences how above coverage criteria, originally defined for imperative languages, can be defined for **ERLANG/EFM**S.

Here **decision** may evaluate to true or false, both evaluations are possible. For example, Decision Coverage would require two test cases: one test case where **statement1** is followed by **statement3** and another test case where **statement2** is followed by **statement3**.

An edge clause with a guard evaluating to false will, by definition, not be included in a test case. Thus, if we let the **decision** above correspond to a guard in an edge clause, only the case where **decision** evaluate to true is possible. Now, it may be the case that evaluating this particular **decision** (i.e., guard) to false may enable the inclusion of some other edge clause. However, such an interpretation would introduce a dependency with the test suite algorithm. Something we do not want, we therefore relax the requirement of inclusion of a false **decision** and consider coverage of a true **decision** "enough" for a test case to be included according to these coverage criteria. In Figure 4.1 this is reflected by removing all tests in the fourth row, in all tables. Thus, Decision Coverage becomes identical to the coverage criteria "all edges of the EFSM" (All-Edges) since covering an edge is equivalent to a true decision, and Condition Coverage

becomes identical to Condition/Decision Coverage, as they only differ in that Condition/Decision Coverage additionally requires test cases such that the decision has one evaluation to false.

Under symbolic execution it may not be known at the time of execution whether a particular condition or decision is true. In Section 3.7 the operational semantics for observers onto an EFSM with symbolic execution of ERLANG/EFSM was given. We there defined the superposition condition that we here might use as an additional imposed condition required for a condition or decision to evaluate to true or false.

To define observers suitable for the coverage criteria above, we define the observer predicates:

Condition Coverage predicate, is_cc/2

Is true iff on an edge the guard has a condition R that is evaluated to the boolean value B. This can be expressed as:

```
is_cc(R,B) ->
  to_bool(LocEnv=lists:zip(EdgeVars,EventPars)) and
  member(R,conds(Guard)) and
  B=eval(R,LocEnv++SourceEnv).
```

Multiple Condition Coverage predicate, is_mcc/1

Is true iff BV is a list of length n with boolean elements, and from the guard on an edge, a list with conditions $[r_1, \dots, r_n]$ sorted in lexical order can be constructed, such that at position i , r_i is evaluated to the boolean value bv_i . This can be expressed as:

```
is_mcc(BV) ->
  to_bool(LocEnv=lists:zip(EdgeVars,EventPars)) and
  to_bool(BV=map({eval,[LocEnv++SourceEnv]},conds(Guard))).
```

Modified Condition/Decision Coverage predicate, is_mcdc/2

Is true iff on an edge the guard has a condition R that is evaluated to the boolean value B, and the evaluation of R affects the evaluation of the guard. This can be expressed as:

```
is_mcdc(R,B) ->
  to_bool(Env=lists:zip(EdgeVars,EventPars) ++ SourceEnv) and
  member(R,conds(Guard)) and
  to_bool(B=eval(R,Env)) and
  (eval(subst(Guard,R,false),Env) /=
   eval(subst(Guard,R,true), Env)).
```

An observer for Condition (and Condition/Decision) Coverage is given in 4.2(i), Multiple Condition Coverage in 4.2(ii), Decision Coverage (and All-Edges) in 4.2(iii) and Modified Condition/Decision Coverage in 4.2(iv).

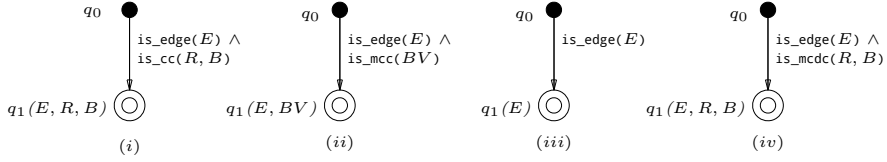


Figure 4.2. Observers for the control flow criteria on guards: (i) Condition (or Condition/Decision) Coverage, (ii) Multiple Condition Coverage, (iii) Decision Coverage (and All-Edges) and (iv) Modified Condition/Decision Coverage.

4.1.2 Coverage of locations

Coverage of locations is a basic, well-known coverage criterion. In an EFSM there are typically only a limited number of locations which also implies that we may expect location coverage only to generate a limited number of different coverage items. A coverage criterion for “*all target locations of the EFSM*” is specified with an observer using **ERLANG/OBS** notation in Example 3.1.

4.1.3 Coverage of paths

A path is a unique sequence of edges from a start location to a stop location.

Examples of coverage criteria on paths includes:

All-paths coverage requires coverage of all paths in the model and is identical to exhaustively searching the complete model and generate a test case for each run between a start location and a stop location. An observer, which collects edges and delivers the covered path as a coverage item, can be defined in **ERLANG/OBS** as

```
-obs_locations({q,[q2],[q1]}).
```

```
q() when is_edge(E), is_startloc() ->
    {next_state,q1([E])}.
```

```
q1(P) when is_edge(E), not is_stoploc() ->
    {next_state,q1([E|P])};
q1(P) when is_edge(E), is_stoploc() ->
    {next_state,q2([E|P])}.
```

Test case length criteria requires coverage of test cases with certain minimum path length. An observer, which collects edges and delivers a coverage item when a stop location is reached after a minimum of n , for $n > 1$, edges have been covered.

```
-obs_locations({q,[q2],[q1]}).
```

```

q() when is_edge(E), is_startloc() ->
    {next_state,q1([E],1)}.

q1(P,K) when is_edge(E), not is_stoploc(), K<5 ->
    {next_state,q1([E|P],K+1)};
q1(P,K) when is_edge(E), K==5 ->
    {next_state,q2([E|P])}.

```

4.1.4 Coverage of data flow

Data flow oriented coverage criteria focus on the data flow part of the specification. In an EFSM this typically involves following runs where state variables are bound to values at some edge and used on some other edge. Usage of state variables can be further divided into *computation-use* if used in the right hand side of an assignment or, *guard-use* if used within a guard [Rapps 85]. For example, in the assignment $A = B + 1$, the state variable A is defined and the state variable B is used. Furthermore, B reaches a computation-use in the assignment $A = B + 1$, but guard-use in the guard $A == B$. A *definition clear path* with respect to B is a sequence of edges where there exists no definition of B . To exemplify a number of common data flow criteria we assume an EFSM in Figure 4.3 with 10 edges e_1, \dots, e_{10} where a state variable is defined in e_1 and e_2 , and used in e_7 and e_{10} . Covered edges are represented with thick arrows.

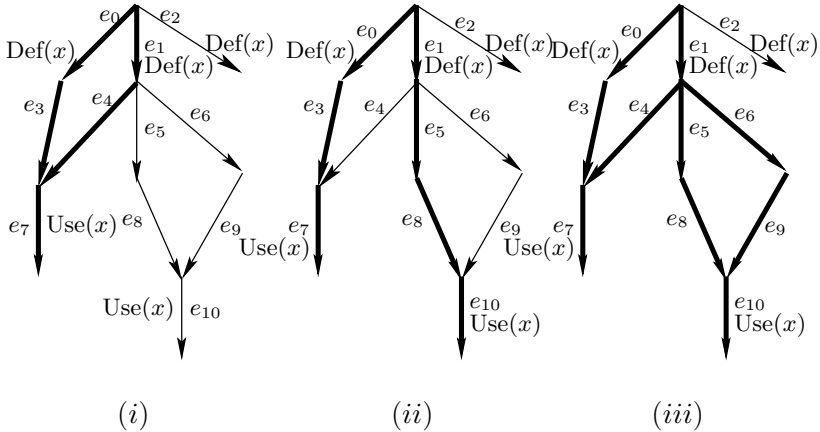


Figure 4.3. EFSM graphs, with annotations for definition and usage of a state variable x , illustrating (i) Reachable definitions coverage, (ii) Reachable uses coverage, and (iii) Definition-Use pair coverage

Reachable definitions coverage [Frankl 88] requires, for a state variable X , coverage of all runs with a definition of X and some definition clear sequence of edges to some usage of X . Thus, the Reachable definitions coverage criterion ensures that all defined state variables will be tested at least once by one of their uses in the model. For example, the two sequences of edges

$$e_0, e_3, e_7 \quad e_1, e_4, e_7$$

in Figure 4.3(i) are sufficient.

Reachable uses coverage [Frankl 88, Clarke 89, Herman 76, Laski 83] requires, for a state variable X , coverage of all usages of X such that there exists a definition clear sequence of edges from where X is defined. Thus, the Reachable uses coverage criterion ensures that *all* uses of state variables previously defined will be tested at least once by one of their uses in the model. For example, the two sequences of edges

$$e_0, e_3, e_7 \quad e_1, e_5, e_8, e_{10}$$

in Figure 4.3(ii) are sufficient.

Definition-Use pair coverage [Frankl 88, Rapps 85] requires, for a state variable X , coverage of all runs where X is defined and there exists a usage of X . Thus, the Definition-Use pair coverage criterion ensures that all possible runs between where a state variable is defined to where the state variable is used will be tested. For example, the three sequences of edges

$$e_0, e_3, e_7 \quad e_1, e_4, e_7 \quad e_1, e_5, e_8, e_{10}$$

in Figure 4.3(iii) are sufficient.

Definition context coverage [Laski 83] requires coverage of all runs such that for every definition of a state variable, every different *definition context* is represented. A definition context is given by all previous assignments of state variables on the current run.

For example, in the EFSM in Figure 4.4 there are definitions of state variables on all edges. Thus, Definition context coverage requires coverage of the four sequences of edges:

$$e_0, e_3, e_4 \quad e_0, e_2, e_3, e_4 \quad e_0, e_1, e_2, e_3, e_4 \quad e_0, e_1, e_3, e_4$$

With the additional requirement of an *ordered definition context*, the sequence of edges e_0, e_2, e_1, e_3, e_4 also needs to be covered. An observer for Ordered definition context coverage is given in Figure 4.6.

Effect pairs coverage [Clarke 89] requires coverage of all runs where a definition of a state variable X is affected by the definition of another

state variable Y . This can be generalized to coverage of state variables affecting the definition of X in k steps. For example, in the EFSM in Figure 4.4 state variable y is affected by the definition of x in two steps since y is bound to the value of z , which is bound to the value of x , which is bound to a value. This is the only run, in this EFSM, where a state variable is given a value that depends on definitions in 2 steps. Thus, Effect pairs coverage requires coverage of the sequence of edges: e_0, e_3, e_4 for $k = 2$. An observer for Effect pairs coverage is given in Figure 4.7.

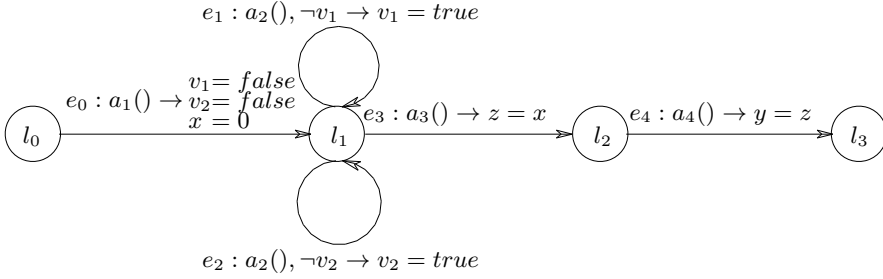


Figure 4.4. EFSM graph example for illustrating Definition context coverage and Effect pairs coverage.

To define observers suitable for the coverage criteria above, we define the observer predicates:

State variable definition predicate, is_definedvar/1

Is true if a state variable is bound with an assignment in the current edge clause. This can be expressed as:

```
is_definedvar(V) ->
    member(V, map(lhs, Assigns)).
```

State variable usage predicate, is_usedvar/1

Is true if a state variable is used (in a guard or assignment) by the current edge clause in the EFSM. This can be expressed as:

```
is_usedvar(U) ->
    member(U, vars(map(rhs, Assigns)) ++
        vars(Guard) ++
        vars(Outexprs) ).
```

State variable directly affected predicate, is_da/2

Is true if, in an assignment, a state variable is defined in terms of some (other) state variable. For example, in an assignment $v_2 = v_1$, v_1 directly affects v_2 . This can be expressed as:

```
is_da(V1,V2) ->
  map({affect,[V1,V2]},Assigns)!=[].
```

where for each assignment A in Assigns , $\text{affect}(A,V1,V2)$ is executed and the result appended. Thus, if there exists an assignment in Assigns where $V1$ directly affects $V2$ the resulting list becomes non-empty. Note that both $V1$ and $V2$ must be bound before usage of $\text{is_da}/2$.

An observer for Reachable definitions coverage is given in Figure 4.5(i), Reachable uses coverage in 4.5(ii), and Definition-Use pair coverage in 4.5(iii).

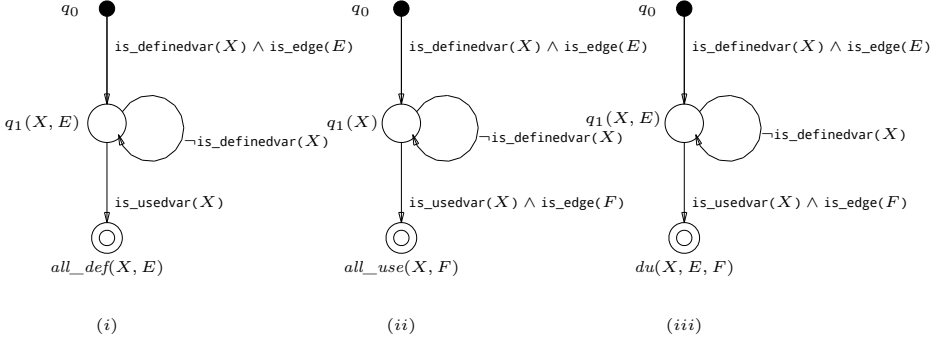


Figure 4.5. Observers for data flow criteria: (i) Reachable definitions coverage, (ii) Reachable uses coverage, and (iii) Definition-Use pair coverage.

4.2 Model-dependent coverage criteria

It is sometimes desirable to create dedicated test suites for a specific model. Typically because some requirement on the system need more thorough testing. Sequence diagrams, showing the message flow, are then often useful to express such test purposes [Grabowski 95]. We will here use Basic Message Sequence Charts (MSC) [ITU-T 99a, Mauw 97] to represent sequence diagrams and show, with an example, how an observer can be constructed (and test suite generated).

A possible sequence of events, represented by an MSC, on the EFSM from Example 1.1 is shown in Figure 4.8. The vertical lines in the MSC are *instances*, i.e., different entities that interact with input and output events. The instances in this MSC are **Me**, **preschool**, **workUU** and **workMA**. The horizontal arrows in the MSC are *messages*, i.e., events where arrows to **Me** represents input events and arrows from **Me** represents output events. We can then map each message in the MSC to a corresponding observer edge in an observer. For example, assume we are interested to generate a test suite from the MSC in Figure 4.9. Let these

```
-obs_locations({q,[def_path],[q1]}).
```

```
q() when not is_stoploc() and  
    is_definedvar(X) and  
    is_edge(E) ->  
    {next_state,q1([E])};  
q() when is_stoploc() and  
    is_definedvar(X) ->  
    {next_state,def_path([])}.
```

```
q1(P) when not is_definedvar(X) ->  
    {next_state,q1(P)};  
q1(P) when is_definedvar(X) and  
    is_edge(E) ->  
    {next_state,def_path([E|P])};  
q1(P) when is_definedvar(X) ->  
    {next_state,def_path(P)}.
```

Figure 4.6. An observer for Ordered definition context coverage defined in ERLANG/OBS.

```
-obs_locations({q,[affect_pair],[q1,q2]}).
```

```
q() when is_definedvar(X) and is_edge(E1) ->  
    {next_state,q1(X,E1)}.
```

```
q1(X,E1) when not is_definedvar(X) ->  
    {next_state,q1(X,E1)};  
q1(X,E1) when is_used(Y) and  
    is_da(Y,X) and  
    is_edge(E1) ->  
    {next_state,q2(X,Y,E1,E2)}.
```

```
q2(X,Y,E1,E2) when not is_definedvar(Y) ->  
    {next_state,q2(X,Y,E1,E2)};  
q2(X,Y,E1,E2) when is_used(Z) and  
    is_da(Z,Y) and  
    is_edge(E3) ->  
    {next_state,affect_pair(X,Z,E1,E2,E3)}.
```

Figure 4.7. An observer for Effect pairs coverage (in 2 steps) defined in ERLANG/OBS.

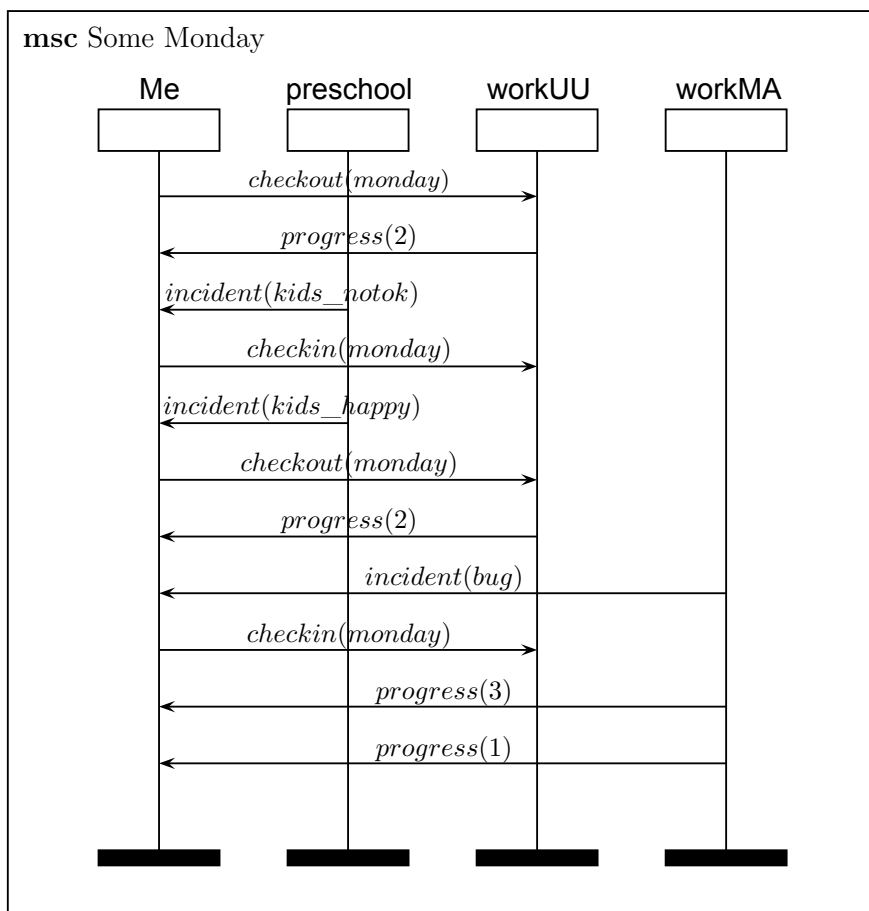


Figure 4.8. A possible sequence of events on the EFSM from Example 1.1.

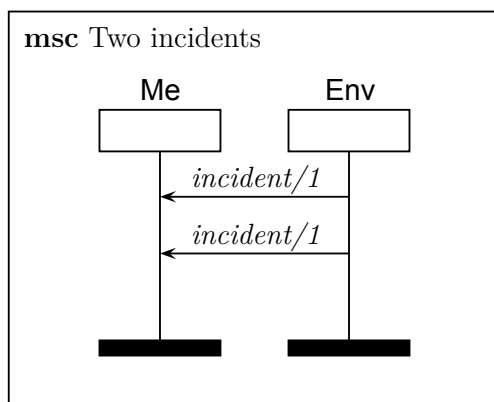


Figure 4.9. A possible MSC that can be used to generate a test suite from.

messages represent a test purpose where all test cases containing at least two consecutive *incident/1* input events should be included in a generated test suite. An observer automaton can then be created as follows.

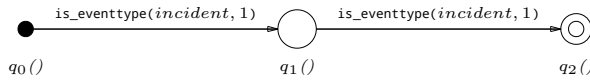


Figure 4.10. An example of an observer for selecting test cases that contains two consecutive incident events.

- We need one new observer state for each input event. Starting with an initial observer state we therefore need two additional observer states.
- Order must be preserved from the MSC, so the observer states will be ordered in a line with the observer predicate `is_eventtype/2` on each observer edge.
- If we want to relax the requirement the input events must be consecutive, a “wait loop” can be added to the observer state between the matching of the *incident/1* events.
- Depending on test cases accepted for inclusion in the test suite, additional observer predicates can be used, and results stored in the observer state. If we just want all possible test cases we can apply the observer automata as a filter, Section 5.5.1.

The resulting observer is shown in Figure 4.10. The MSC in Figure 4.8 represents one example of such a test case.

5. Generating test suites

This chapter presents our technique for generating test suites (i.e., sets of test cases) from **ERLANG/EFSM** models extended with observers.

Recall that the formal model of the SUT represents an abstraction of the interaction between the SUT and its environment. It follows that the test suites generated from the model will be at the same level of abstraction. We therefore use the term *abstract test case* for test cases that are at the same level of abstraction as the model. Note that the definition of abstract test case is identical to the definition of a trace of an EFSM. To emphasize that (ordinary) test cases, which are the level of abstraction of the SUT, are different from abstract test cases, we sometimes call them *concrete test cases*. Concrete test cases can be generated from abstract test cases by translating events in the abstract test case to events that can actually be received and transmitted by the SUT.

Since abstract test cases correspond to executions of the **ERLANG/EFSM** model, we can in principle generate abstract test suites (i.e., sets of abstract test cases) by enumerating these executions. However, this approach is not practically feasible. We therefore use a symbolic approach, which generates *symbolic test cases*, as defined in Definition 2.9, from an **ERLANG/EFSM** model. A symbolic test case is a compact representation of a set of abstract test cases of the EFSM. In a symbolic test case, data in input events is represented symbolically by parameters; the range of actual values of the parameters is constrained by constraints. A symbolic test case represents the set of abstract test cases that can be obtained by instantiating parameters by values that satisfy these constraints.

This chapter explains how symbolic test suites are generated, using the observer automata introduced in Chapter 3, and how they can be transformed into a format suitable for testing of a concrete implementation of a SUT.

This chapter is organized as follows. Section 5.1 introduces a symbolic combination of EFSM and observer runs, and shows how to use it to extract test cases. In Section 5.2, this representation is used to derive a symbolic test suite generation algorithm. Section 5.3 introduces an efficient representation of observer states in the implementation of the test suite generation algorithm. Section 5.4 presents our techniques for mapping abstract test cases, derived from the model, to concrete test cases, which can be applied on the system under test. Section 5.5 describes alternative usages of **ERLANG/OBS**.

5.1 Generating Symbolic Test Cases

Let us consider how we can generate an abstract test suite, i.e., a set of abstract test cases, to cover a given set of coverage items. From Theorem 3.17 we know that test cases that cover a coverage item q_f can be obtained from a symbolic test case of the EFSM, which symbolically covers an accepting conditioned observer state $\langle q_f, H_n \rangle$ for some H_n . A test case that covers q_f can be obtained by instantiating symbolic parameters in the symbolic test case such that the path condition G_n and the superposition condition H_n are satisfied.

In order to search for a symbolic test case which symbolically covers $\langle q_f, H_n \rangle$, we need to generate symbolic runs of both the EFSM and the observer, which match. The symbolic run of the EFSM

$$\langle l_0, \bullet, \text{true} \rangle \xrightarrow{a_1(\overline{p_1})/\overline{b_1}} \dots \xrightarrow{a_n(\overline{p_n})/\overline{b_n}} \langle l_n, \sigma_n, G_n \rangle$$

should be such that there is symbolic run of the observer

$$\langle q_0, \text{true} \rangle \xrightarrow{\zeta_1^s} \dots \xrightarrow{\zeta_n^s} \langle q_n, H_n \rangle$$

over the corresponding sequence of symbolic annotated computation steps. Furthermore, since observers are nondeterministic, each run of an EFSM can in general correspond to a set of observer runs. The close correspondence between (symbolic) EFSM runs and sets of (symbolic) observer runs suggest that the test suite generation should generate them together. We therefore introduce a new kind of state, which combines of a set of (symbolic) observer states with a (symbolic) EFSM state. We will say that the observer states are *superposed* onto the EFSM states. Furthermore, the symbolic trace should be remembered, so that the generated symbolic test case can be extracted from successful explorations. This motivates the following definition.

Definition 5.1 A *symbolic superpositioned state* is a triple of form

$$\langle \langle l, \sigma, G \rangle \parallel \mathcal{Q} \parallel \omega \rangle$$

where $\langle l, \sigma, G \rangle$ is a symbolic state, \mathcal{Q} a set of conditioned observer states (see Section 3.8), and ω a symbolic trace which leads to the symbolic state $\langle l, \sigma, G \rangle$. \square

Definition 5.2 A *symbolic superpositioned computation step* is a triple

$$\langle \langle l, \sigma, G \rangle \parallel \mathcal{Q} \parallel \omega \rangle \xrightarrow{a(\overline{p})/\overline{b}} \langle \langle l', \sigma', G' \rangle \parallel \mathcal{Q}' \parallel \omega \cdot a(\overline{p})/\overline{b} \rangle$$

where

- $\langle l, \sigma, G \rangle \xrightarrow{a(\overline{p})/\overline{b}} \langle l', \sigma', G' \rangle$ is a symbolic computation step; we let ς^s represent the corresponding symbolic annotated computation step,
- $\mathcal{Q}' = \left\{ \langle q', H' \rangle \left| \begin{array}{l} \langle q, H \rangle \xrightarrow{\varsigma^s} \langle q', H' \rangle \\ \langle q, H \rangle \in \mathcal{Q} \end{array} \right. \right\}$ is the set of conditioned observer states to which the observer can make conditioned observer steps from some conditioned observer state $\langle q, H \rangle$ in \mathcal{Q} , triggered by ς^s , and
- ω is a symbolic trace ω . □

Definition 5.3 A *symbolic superposition run* over a symbolic trace is a sequence of symbolic superpositioned computation steps

$$\langle \langle l_0, \bullet, \text{true} \rangle \parallel \{ \langle q_0, \text{true} \rangle \} \parallel \epsilon \rangle \xrightarrow{a_1(\overline{p_1})/\overline{b_1}} \dots \xrightarrow{a_n(\overline{p_n})/\overline{b_n}} \langle \langle l_n, \sigma_n, G_n \rangle \parallel \mathcal{Q}_n \parallel \omega_n \rangle$$

labeled by the input-output event expressions of the symbolic trace. □

We note that by the assumption that observers have self-loops from initial and accepting locations, stated in Section 3.1, the set \mathcal{Q} of conditioned observed states will always contain the initial observer state q_0 , and that once an accepting observer state has entered \mathcal{Q} , it will remain in \mathcal{Q} .

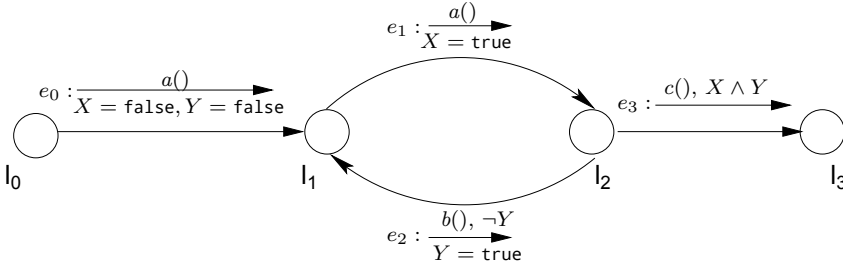


Figure 5.1. A graphical representation of a simple EFMSM with a unique name (e_x) associated to each edge.

Example 5.4 If the observer for Definition-Use pair coverage (see Figure 4.5(iii) in Section 4.1.4) is superposed onto the EFSM in Figure 5.1, the following symbolic superposition run can be taken

$$\begin{aligned}
& \langle \langle l_0, \bullet, \text{true} \rangle \parallel \{ \langle q_0, \text{true} \rangle \} \parallel \epsilon \rangle \xrightarrow{a()/} \\
& \langle \langle l_1, \left\{ \begin{array}{l} X := \text{false} \\ Y := \text{false} \end{array} \right\}, \text{true} \rangle \parallel \left\{ \begin{array}{l} \langle q_0, \text{true} \rangle \\ \langle q_1('X', e_0), \text{true} \rangle \\ \langle q_1('Y', e_0), \text{true} \rangle \end{array} \right\} \parallel a()/ \rangle \xrightarrow{a()/} \\
& \langle \langle l_2, \left\{ \begin{array}{l} X := \text{true} \\ Y := \text{false} \end{array} \right\}, \text{true} \rangle \parallel \left\{ \begin{array}{l} \langle q_0, \text{true} \rangle \\ \langle q_1('X', e_1), \text{true} \rangle \\ \langle q_1('Y', e_0), \text{true} \rangle \end{array} \right\} \parallel a()/ \cdot a()/ \rangle \xrightarrow{b()/} \\
& \langle \langle l_1, \left\{ \begin{array}{l} X := \text{true} \\ Y := \text{true} \end{array} \right\}, \text{true} \rangle \parallel \left\{ \begin{array}{l} \langle q_0, \text{true} \rangle \\ \langle q_1('X', e_1), \text{true} \rangle \\ \langle q_1('Y', e_2), \text{true} \rangle \\ \langle du('Y', e_0, e_2), \text{true} \rangle \end{array} \right\} \parallel a()/ \cdot a()/ \cdot b()/ \rangle \xrightarrow{a()/} \\
& \langle \langle l_2, \left\{ \begin{array}{l} X := \text{true} \\ Y := \text{true} \end{array} \right\}, \text{true} \rangle \parallel \left\{ \begin{array}{l} \langle q_0, \text{true} \rangle \\ \langle q_1('X', e_1), \text{true} \rangle \\ \langle q_1('Y', e_2), \text{true} \rangle \\ \langle du('Y', e_0, e_2), \text{true} \rangle \end{array} \right\} \parallel a()/ \cdot a()/ \cdot b()/ \cdot a()/ \rangle \xrightarrow{c()/} \\
& \langle \langle l_3, \left\{ \begin{array}{l} X := \text{true} \\ Y := \text{true} \end{array} \right\}, \text{true} \rangle \parallel \left\{ \begin{array}{l} \langle q_0, \text{true} \rangle \\ \langle du('Y', e_0, e_2), \text{true} \rangle \\ \langle du('X', e_1, e_3), \text{true} \rangle \\ \langle du('Y', e_2, e_3), \text{true} \rangle \end{array} \right\} \parallel a()/ \cdot a()/ \cdot b()/ \cdot a()/ \cdot c()/ \rangle
\end{aligned}$$

Note that this single symbolic run of the EFSM can cover several accepting symbolic observer states. For example, the three possible definition-use pairs represented by the accepting symbolic observer states $du('Y', e_0, e_2)$, $du('X', e_1, e_3)$, and $du('Y', e_2, e_3)$ are covered by a single symbolic superposition run. \square

In Theorem 3.17 we established a correspondence between symbolic coverage and (non-symbolic) coverage. It follows from the definition of symbolic superposition run that if

$$\langle \langle l_0, \bullet, \text{true} \rangle \parallel \{ \langle q_0, \text{true} \rangle \} \parallel \epsilon \rangle \xrightarrow{a_1(\overline{p_1})/\overline{b_1}} \dots \xrightarrow{a_n(\overline{p_n})/\overline{b_n}} \langle \langle l_n, \sigma_n, G_n \rangle \parallel \mathcal{Q}_n \parallel \omega_n \rangle$$

is a symbolic superposition run, then the symbolic run

$$\langle l_0, \bullet, \text{true} \rangle \xrightarrow{a_1(\overline{p_1})/\overline{b_1}} \dots \xrightarrow{a_n(\overline{p_n})/\overline{b_n}} \langle l_n, \sigma_n, G_n \rangle$$

of the EFSM symbolically covers each accepting conditioned observer state in \mathcal{Q}_n . Each covered conditioned observer state, which is of the

form $\langle \iota(\bar{e}), H \rangle$, represents the set of coverage items $\iota(\bar{d})$ that can be obtained by finding a parameter environment so that \bar{e} become values in the appropriate domains, and so that G_n and H_n are satisfied. Thus, a test case that covers $\iota(\bar{d})$ can be obtained from the symbolic trace w by instantiating the parameter environment so that $(G_n \wedge H_n \wedge \bar{e} = \bar{d})$ is satisfied.

We will generate test suites by systematically exploring symbolic superposition runs, and extracting a set of symbolic test cases, so that each coverage item is covered by at least one symbolic test case in the symbolic test suite. The previous paragraph suggests that from each symbolic superposition run

$$\langle \langle l_0, \bullet, \text{true} \rangle \parallel \{ \langle q_0, \text{true} \rangle \} \parallel \epsilon \rangle \xrightarrow{a_1(\bar{p}_1)/\bar{b}_1} \dots \xrightarrow{a_n(\bar{p}_n)/\bar{b}_n} \langle \langle l_n, \sigma_n, G_n \rangle \parallel \mathcal{Q}_n \parallel \omega_n \rangle,$$

we should extract the set of symbolic test cases

$$\langle w, G_n \wedge H_n \wedge \bar{e} = \bar{d} \rangle$$

such that there is some $\langle \iota(\bar{e}), H \rangle \in \mathcal{Q}_n$ for which the coverage item $\iota(\bar{d})$ has not yet been covered, and such that there is some parameter environment that can satisfy $G_n \wedge H_n \wedge \bar{e} = \bar{d}$.

5.2 Generating symbolic test suites

The problem of generating a symbolic test suite can be addressed by symbolic state space exploration, which searches for accepting states in a symbolic state space. A symbolic state space exploration algorithm to compute a symbolic test suite is shown in Figure 5.2. The algorithm assumes an EFSM specified in `ERLANG/EFsm` and an observer specified in `ERLANG/Obs`. It computes a set of symbolic test cases that cover some set of coverage items of the observer. The set of covered coverage items can be controlled by appropriate instantiation of the function *extract_tcs* and predicate *stopcond*.

In the algorithm of Figure 5.2

- `Wait` contains the symbolic superpositioned states waiting to be explored,
- `Cov` is the set of already covered coverage items,
- `TS` is the current set of generated symbolic test cases,
- `Lstop` is the set of stop locations in the EFSM, and
- *extract_tcs* is a function, which depends on the set `Cov` of already covered coverage items, which extracts new symbolic test cases from a symbolic superposition state when the exploration reaches the end of a symbolic run of the EFSM. We elaborate on *extract_tcs* in more detail below.

- *stopcond* is a stop condition that determines when no more symbolic test cases should be created and the test suite is considered complete. A condition for not stopping until the complete state space has been searched is $\text{WAIT} = \emptyset$.

```

Cov :=  $\emptyset$ , TS :=  $\emptyset$ ,
WAIT :=  $\{\langle l_0, \bullet, \text{true} \rangle \parallel \{\langle q_0, \text{true} \rangle\} \parallel \epsilon\}$ 
while  $\neg \text{stopcond}$  do
  select_and_remove  $\langle l, \sigma, G \rangle \parallel \mathcal{Q} \parallel \omega$  from WAIT
  if  $l \notin L^{\text{stop}}$  then
    for all  $\langle l, \sigma, G \rangle \parallel \mathcal{Q} \parallel \omega \xrightarrow{a(\bar{p})/\bar{b}} \langle l', \sigma', G' \rangle \parallel \mathcal{Q}' \parallel \omega \cdot a(\bar{p})/\bar{b}$ 
      add  $\langle l', \sigma', G' \rangle \parallel \mathcal{Q}' \parallel \omega \cdot a(\bar{p})/\bar{b}$  to WAIT
    else
      TS := extract_tcs(Cov)  $\cup$  TS
      Cov := {coverage items covered by extract_tcs(Cov)}  $\cup$  Cov
  return TS

```

Figure 5.2. A symbolic state space exploration algorithm for test suite generation.

Initially, the set of already explored states is empty and the only state waiting to be explored is $\langle l_0, \bullet, \text{true} \rangle \parallel \{\langle q_0, \text{true} \rangle\} \parallel \epsilon$. The algorithm then repeatedly selects and removes a symbolic superpositioned state $\langle l, \sigma, G \rangle \parallel \mathcal{Q} \parallel \omega$ from WAIT and examines it. If the selected state is not a stop location (i.e., it has outgoing edges), then all successor symbolic superpositioned states reachable from $\langle l, \sigma, G \rangle \parallel \mathcal{Q} \parallel \omega$ via a symbolic superposition step are inserted into WAIT. When reaching a stop location, new test cases are extracted using *extract_tcs*(Cov). The algorithm terminates when the condition *stopcond* becomes true.

Let us now elaborate on the function *extract_tcs*(Cov) in more detail. In order to generate test cases that cover as many coverage items of the observer as possible, the function *extract_tcs*(Cov) should extract test cases whenever they cover coverage items that are not already in Cov. By the discussion in the previous subsection, *extract_tcs*(Cov) should extract the set of symbolic test cases

$$\left\{ \langle w, G \wedge H \wedge \bar{e} = \bar{d} \rangle \mid \begin{array}{l} \exists \langle \iota(\bar{e}), H \rangle \in \mathcal{Q} \cdot \iota(\bar{d}) \notin \text{Cov} \\ \wedge \exists \Gamma \cdot \Gamma \vdash (G \wedge H \wedge \bar{e} = \bar{d}) \end{array} \right\}$$

Intuitively, this is the set of symbolic test cases of form

$$\langle w, G \wedge H \wedge \bar{e} = \bar{d} \rangle$$

such that there is some $\langle \iota(\bar{e}), H \rangle \in \mathcal{Q}$ for which the coverage item $\iota(\bar{d})$ has not yet been covered, and can be obtained by using a parameter environment that satisfies $G \wedge H \wedge \bar{e} = \bar{d}$.

A very common case is that the test generation algorithm generates only conditioned observer states $\langle \iota(\bar{e}), H \rangle$ that do not have any symbolic parameters, and where H is **true**. That is, they are of form $\langle \iota(\bar{d}), \text{true} \rangle$. In which case we can simply represent them as nonsymbolic coverage items $\iota(\bar{d})$. In this case, $\text{extract_tcs}(\text{Cov})$ can be simplified to

$$\{ \langle w, G \rangle \mid \langle \iota(\bar{d}), \text{true} \rangle \in \mathcal{Q} \wedge \iota(\bar{d}) \notin \text{Cov} \},$$

i.e., it need only check whether \mathcal{Q} contains any coverage items that are not already in Cov . In the rest of this thesis we will only consider this simpler case, i.e., when symbolic observer states do *not* contain symbolic parameters.

5.2.1 Refining the search exploration algorithm

The search algorithm affects the order in which coverage items are covered and therefore influences size of test suite (and test cases) and speed with which symbolic test suites are generated. For example, a breadth-first algorithm is guaranteed to generate a test suite with the shortest test cases but can be expected to consume much memory. A depth-first algorithm consumes less memory, but a biased left-to right search will cause a correspondingly biased exploration of the state space. In [Pretschner 01] a comparison between different search algorithms on experimental data in a Constraint Logic Programming (CLP) framework is performed. Pretschner concludes that in general a depth-first algorithm can be expected to perform best when generating a test suite.

Many coverage criteria (e.g., Def-Use) benefit to a large degree from a search algorithm which distributes symbolic test cases over the entire EFSM, resulting in less needed test cases. This implies that biased left-to right search often is not optimal and that a random selection may be expected to perform better. Still better, if possible, is to use some algorithm to efficiently approximate a best-first selection.

5.3 Efficiently representing sets of states

A problem in test suite generation algorithms is to efficiently represent and manipulate the sets \mathcal{Q} and Cov , of conditioned observer states of an observer superposed onto an EFSM. In this section, we show how these sets can be efficiently represented and manipulated, using techniques inspired by bitvector analysis [Knoop 96], which is often applied in e.g., data-flow analysis.

Recall that we henceforth only considers the case when conditioned observer states do not depend on symbolic parameters and that L^o is the set of observer locations. For an observer location $\iota \in L^o$, let D_1, \dots, D_m be the domains of the parameters d_1, \dots, d_m in observer states of form $\iota(d_1, \dots, d_m)$. A set \mathcal{Q} of conditioned observer states can then be represented as a mapping

$$\mathcal{Q} : L^o \rightarrow (D_1 \times \dots \times D_m \rightarrow [0, 1])$$

such that for each observer location ι , we have $\mathcal{Q}(\iota)(d_1, \dots, d_m) = 1$ if $\langle \iota(d_1, \dots, d_m), \text{true} \rangle \in \mathcal{Q}$, otherwise $\mathcal{Q}(\iota)(d_1, \dots, d_m) = 0$. In a symbolic superpositioned computation step of form

$$\langle \langle l, \sigma, G \rangle \parallel Q \parallel \omega \rangle \xrightarrow{a(\bar{p})/\bar{b}} \langle \langle l', \sigma', G' \rangle \parallel Q' \parallel \omega \cdot a(\bar{p})/\bar{b} \rangle$$

this mapping is then, letting ς^s denote the current symbolic annotated computation step, updated according to

$$\mathcal{Q}'(\iota')(d'_1, \dots, d'_{m'}) := \bigvee_{\langle \iota(d_1, \dots, d_m), \text{true} \rangle \xrightarrow{\varsigma^s} \langle \iota'(d'_1, \dots, d'_{m'}), \text{true} \rangle} \mathcal{Q}(\iota)(d_1, \dots, d_m) \quad ,$$

where \mathcal{Q}' is the updated set of conditioned observer states, and the disjunction is taken over all conditioned observer states from which there is a conditioned observer step labeled by ς^s .

The above definition of how the mappings \mathcal{Q} are updated are abstract, formulated using the semantic definition of the symbolic observer step relation. We can make them more concrete by replacing the definition by the corresponding premises in the symbolic operational semantics for observer edge clauses in Section 3.7.3. First recall that the definition of the conditioned observer step relation, from Section 3.8, of form

$$\langle \iota(d_1, \dots, d_m), \text{true} \rangle \xrightarrow{\varsigma^s} \langle \iota'(d'_1, \dots, d'_{m'}), \text{true} \rangle$$

implies that for some ς^s there is a transition in the operational semantics of form

$$\langle \iota(d_1, \dots, d_m), \varsigma^s, \bullet \rangle \Rightarrow \langle \{\text{next_state}, \iota'(d'_1, \dots, d'_{m'})\}, \text{true}, \epsilon, \varsigma^s, \rho \rangle \quad ,$$

which happens only if there exists an observer edge clause of form

$$\iota(u_1, \dots, u_m) \text{ when } h \rightarrow oce$$

such that d_1, \dots, d_m match the patterns u_1, \dots, u_m , the observer guard h is satisfied by the symbolic annotated computation step, represented by ς^s , and $\{\text{next_state}, \iota'(d'_1, \dots, d'_{m'})\}$ is the resulting value of evaluating

oce. We can summarize this discussion by rewriting the rule for updating the set \mathcal{Q} as

$$\begin{aligned} \mathcal{Q}'(\iota')(d'_1, \dots, d'_{m'}) &:= \bigvee \mathcal{Q}(\iota)(d_1, \dots, d_m) \ , \\ &\quad \iota(u_1, \dots, u_m) \text{ when } h \rightarrow oce \\ &\quad \text{match}(u_1, \dots, u_m, d_1, \dots, d_m, \bullet, \rho') \\ &\quad \langle h, \varsigma^s, \rho' \rangle \Rightarrow \langle \text{true}, \epsilon, \varsigma^s, \rho'' \rangle \\ &\quad \langle oce, \varsigma^s, \rho'' \rangle \Rightarrow \langle \text{next_state}, \iota'(d'_1, \dots, d'_{m'}), \epsilon, \varsigma^s, \rho''' \rangle \end{aligned}$$

Example 5.5 Lets take a closer look at how the observer states in the Definition-Use pair observer (Section 4.1.4) are affected by a conditioned observer step. Let V be the set of state variables and E the set of edges in an EFSM. We then have that,

$$\begin{aligned} &\{ \langle q_0(), \text{true} \rangle \} \\ \cup &\{ \langle q_1(x, e), \text{true} \rangle \mid x \in V \wedge e \in E \} \\ \cup &\{ \langle du(x, e, f), \text{true} \rangle \mid x \in V \wedge e, f \in E \} \end{aligned}$$

is the set of conditioned observer states for the observer. As the observer has three locations, we maintain $\mathcal{Q}(q_0)$, $\mathcal{Q}(q_q)$, and $\mathcal{Q}(du)$, which are updated according to the following definitions.

$$\mathcal{Q}(q_0) := 1 \tag{5.1}$$

$$\mathcal{Q}'(q_1)(x, e) := \left(\begin{array}{c} \left(\begin{array}{c} \mathcal{Q}(q_0) = 1 \\ \wedge \text{is_definedvar}(x) \\ \wedge \text{is_edge}(e) \end{array} \right) \\ \vee \left(\begin{array}{c} \mathcal{Q}(q_1)(x, e) = 1 \\ \wedge \neg \text{is_definedvar}(x) \end{array} \right) \end{array} \right) \tag{5.2}$$

$$\mathcal{Q}'(du)(x, e, f) := \left(\begin{array}{c} \left(\begin{array}{c} \mathcal{Q}(q_1)(x, e) = 1 \\ \wedge \text{is_usedvar}(x) \\ \wedge \text{is_edge}(e) \end{array} \right) \\ \vee \left(\begin{array}{c} \mathcal{Q}(du)(x, e, f) = 1 \\ \wedge \text{true} \end{array} \right) \end{array} \right) \tag{5.3}$$

where \mathcal{Q}' represents the set of observer states \mathcal{Q} *after* all updates required by the observer have been performed. Note that the mapping $\mathcal{Q}(q_0)$ of the initial observer location is always 1 (as described by Definition 5.1) and that a bit in the bitvector $\mathcal{Q}(du)$, representing an accepting symbolic observer state, will remain 1 once it becomes 1, as stated by the last disjunct of Definition 5.3. \square

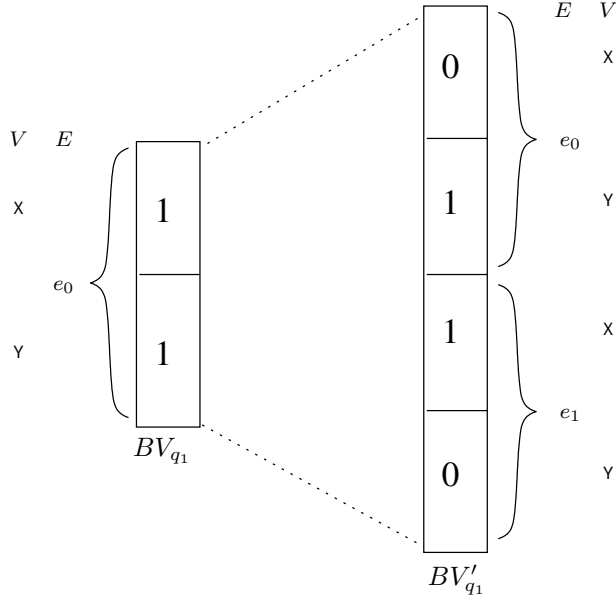


Figure 5.3. The bitvector BV_{q_1} represents the set of observer states $\{q_1(X, e_0), q_1(Y, e_0)\}$ before a conditioned observer step. As the EFSM holds a definition of a variable X on a (new) edge e_1 , the size of the bitvector is adjusted, before Definition 5.2 (from Example 5.5) is applied. The bitvector BV'_{q_1} , after the conditioned observer step, then represents the set of observer states $\{q_1(Y, e_0), q_1(X, e_1)\}$.

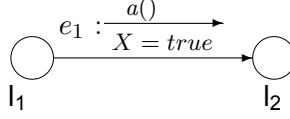
5.3.1 Bitvector Representation of Sets of Observer States

In our implementation, each mapping $\mathcal{Q}(\iota)$ is represented by a one-dimensional bitvector $BV_\iota : [1, \dots, \mathfrak{M}_\iota] \rightarrow [0, 1]$, where $\mathfrak{M}_\iota = \prod_{1 \leq i \leq n} |D_i|$. By defining a bijection $f_\iota : (D_1 \times \dots \times D_m) \rightarrow \mathfrak{M}_\iota$, we can represent $\mathcal{Q}(\iota)(d_1, \dots, d_m)$ as $BV_\iota(f_\iota(d_1, \dots, d_m))$. Since the size of the domains for d_1, \dots, d_m may not be known in advance we allow \mathfrak{M}_ι to increase dynamically when required.

As we assume the observer to have a finite set of observer states, each parameter must have a finite domain. Parameter values in observer states origin from observer predicates. In Section 3.3 we described how definitions of observer predicates includes any valid **ERLANG** expression. which implies that, in general, the size of the domain may be unknown when symbolic test case generation starts. Thus, a bitvector BV_ι representing an observer location, and the associated bijection $f_\iota(d_1, \dots, d_m)$, may need to be dynamically adjusted after a conditioned observer step.

To keep the function f_ι simple the complete bitvector can be reorganized whenever the (known) size of a domain of an observer parameter is adjusted.

Example 5.6 Recall Example 5.4, with a Definition-Use pair observer, and consider the edge



in the EFSM. Assume the set of edges in the EFSM is unknown before evaluation starts. Let the bitvector BV_{q_1} represent the set of observer states $\{q_1(x, e_0), q_1(y, e_0)\}$. Now consider a conditioned observer step over the above edge in the EFSM, see Figure 5.3. The edge e_1 from the EFSM, is bound to the match variables, representing the current symbolic annotated computation step. As this edge is previously not represented in the bitvector, the size of the bitvector BV_{q_1} is adjusted. On the adjusted bitvector we may then apply Definition (5.2) from Example 5.5. The resulting bitvector BV'_{q_1} , after the conditioned observer step, then represents the set of observer states $\{q_1(y, e_0), q_1(x, e_1)\}$. In Example 5.4 both BV_{q_1} and BV_{du} may grow dynamically in size. After the symbolic superposition run the bitvectors are of size: $\mathfrak{M}_{q_0} = 1$, $\mathfrak{M}_{q_1} = 2 \times 3 = 6$, and $\mathfrak{M}_{du} = 2 \times 3 \times 2 = 12$. \square

5.4 Concretisation

In Section 5.2 we presented a method for generating symbolic test suites from an EFSM and an observer. We must perform the following two operations in order to obtain a test suite that can be applied directly on the SUT.

- Each symbolic test case in the symbolic test suite represents a set of abstract test cases. These abstract test cases can be obtained by instantiating the parameters by actual values so that the path condition is satisfied.
- As outlined in Section 1.5.4, the events in an abstract test case are abstractions of events that can actually be exchanged between the SUT and its environment. In order to obtain (concrete) test cases, we must therefore map the events in the abstract test case to corresponding concrete events. In general the domains of the concrete events are much larger than the corresponding domains of the abstract events. We must therefore carefully select concrete counterparts.

In Section 5.4.1 we describe how an abstract test suite can be generated from a symbolic test suite. In Section 5.4.2 we describe how a concrete test suite, suitable for execution against a SUT, can be generated from an abstract test suite.

5.4.1 Generating abstract test suites from symbolic test suites

Recall that a symbolic test suite is a set of symbolic test cases $\langle w, G \wedge H \rangle$ where the path condition G and superposition condition H are boolean expressions over symbolic parameters. Let $\bar{\phi}$ denote all symbolic parameters. Also recall that an abstract test case is simply a trace. An abstract test case can thus be obtained from $\langle w, G \wedge H \rangle$ by choosing an assignment of values to the symbolic parameters $\bar{\phi}$ such that $G \wedge H$ evaluates to **true**. In general, there are many possible such assignments for each symbolic test case. Finding them can be formulated as a constraint satisfaction problem [Gotlieb 98]. Let us illustrate by an hypothetical example

Example 5.7 Consider a symbolic test case

$$\left\langle \begin{array}{l} slir(MS, MlpAge)/ati(MS) \\ ati_resp(Age, Pos)/slia(ok, Pos) \end{array}, G \right\rangle$$

where G is $Age + Clock \leq MlpAge \wedge Clock \leq 3 \wedge Age \leq 3$. This test case starts by a user sending a *slir* event to a SUT requesting the position of subscriber *MS* with a maximum age of *MlpAge*. On this request the SUT starts a clock *Clock* and performs an *ati* request towards the network. In response, the SUT receives an *ati_resp* with a position *Pos* and *Age* of that position. Now, this is an *ok* position if the path condition $Age + Clock \leq MlpAge \wedge Clock \leq 3 \wedge Age \leq 3$ is true. If we assume *Clock* has a domain $\{0, 1, 2\}$ and both *Age* and *MlpAge* have domains $\{1, 2, 3, 4, 5\}$, a constraint solver will be able to find the 10 possible solutions:

	<i>MlpAge</i>	<i>Clock</i>	<i>Age</i>
$\overline{d_1} =$	1	0	1
$\overline{d_2} =$	2	0	2
$\overline{d_3} =$	2	≤ 1	1
$\overline{d_4} =$	3	0	3
$\overline{d_5} =$	3	≤ 1	2
$\overline{d_6} =$	3	≤ 2	1

□

In the evaluation in Section 7 the generated constraints are on a simple form, e.g., $MlpAge = 3$, with no dependencies between different symbolic parameters. Thus, after rewriting $G \wedge H$ into a disjunctive normal form each disjunct represents the necessary conditions for a distinctive set of possible instantiations.

As a selected symbolic test case may be instantiated in multiple ways we may want to only select a few to be executed against the SUT. We therefore define *abstract test case selection techniques* to only select a few of many possible abstract test cases that can be generated from a symbolic test case. Examples of such selections include:

- **Biased** Create a lexicographic ordering of all possible abstract test cases from a symbolic test case and pick the first one. This strategy will naturally include abstract test cases with symbolic parameters biased to some specific values, due to the ordering. Included here for reference only.
- **Rndsym** Select one random abstract test case, out of all possible from a symbolic test case.

In the evaluation, Section 7, we have some experimental results and discussion on usage of abstract test case selection techniques.

5.4.2 Generating concrete test suites from abstract test suites

A SUT requires each event sent to the SUT, by some environment, to be in a representation the SUT can understand. In this section we outline how a generated abstract test suite, see Section 5.4.1, can be used when generating a test suite for execution against a SUT.

We say that events on a format accepted as input by the SUT and emitted as output from the SUT are *concrete events*. We will not further define concrete events, but simply note that concrete events, part of a concrete test case, are created from an abstract test case via some *concretization*. There may be many possible concretizations of an abstract test case to a concrete test case. Further, a concrete test case may not only depend on the abstract test case as there can be implicit constraints on the concretization (e.g., hardware in the SUT, or location of the SUT). Thus, in general, to find a concrete test case can be formulated as a constraint satisfaction problem, similar as for an abstract test case in Section 5.4.1.

To validate if output sent from the SUT conforms to an output event $b(\bar{d})$ in an abstract test case we must consider all concretizations of $b(\bar{d})$. We do this by an *abstraction* of the output sent from the SUT to $b(\bar{d})$. When validating execution of a test case against SUT via some test environment we end up with concretisations/abstractions as outlined in Figure 5.4.

It can be noted that both concretisation and abstraction may depend on the abstract test case. Thus, for example, in Figure 5.4, $a_1(\bar{d}_1)$ and $a_2(\bar{d}_2)$ may be exactly identical abstract events (i.e., $a_1 = a_2$ and $\bar{d}_1 = \bar{d}_2$), but have different concretizations. See a typical simplified use case in Figure 7.3 from the case study in Section 7.1.1 where, different concretiza-

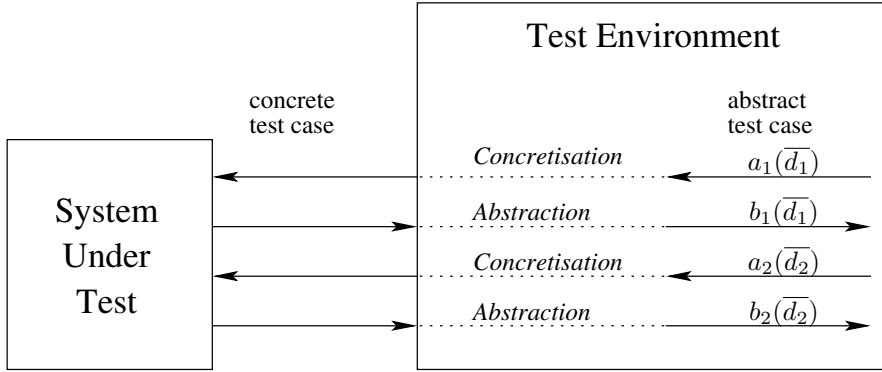


Figure 5.4. Outline of how input events in an abstract test cases are made concrete before sent to the SUT, and responses from the SUT made abstract before validating.

tions of the input event $atir/4$, from the HLR, is needed to distinguish the origin of presence information sent to the Application.

5.5 Alternative usage of observer automata

Observer automata express properties of traces in the EFSM. In the previous section we used observer automata, superposed onto an EFSM, to generate test suites. For inclusion, it was necessary for each test case to reach at least one accepting observer state, not previously reached by any other test case. However, this is just one of several possible usages of observer automata superposed onto an EFSM.

Below we will further discuss usage of observer automata considered of particular interest, *filter* usage in Section 5.5.1 and *property validation* in Section 5.5.2. Declaration of how defined observer automata should be used was explained in Section 3.2.1.

5.5.1 Filtering the specification

A filter is a usage of the observer automata where a test case is accepted for inclusion in a test suite whenever at least one accepting (symbolic) observer state is reached when the EFSM reaches a stop location. Filters are useful for generating a test suites that covers a certain part or fulfills a certain property of a specification. One can think of using observer automata as filter, as projecting the executions of the EFSM to cover executions that satisfy some requirements. For example, all test cases where some state variable or some input event is used, or all test cases

where some model-dependent behavior can be expressed with an MSC, see Section 4.2.

For the case where conditioned observer states do not depend on symbolic parameters, the algorithm in Figure 5.2 can be modified to support filter usage using *extract_tcs* defined by can be simplified to be

$$\{\langle w, G \rangle \mid \langle \iota_f(\bar{d}), \text{true} \rangle \in \mathcal{Q}\}.$$

Further, as we not keep track of coverage items, *Cov* can be removed.

Usage of observer automata as a filter is declared with an observer usage declaration set to *filter*.

Example 5.8 In a filter specification

```
-obs_locations(used_var,[anything],[]).
-obs_usage(used_var,filter).
```

```
used_var() when is_usedvar('Progress') ->
    {next_state,anything()}.
```

we declare observer usage *filter* and an initial observer location *used_var*. As before, the observer predicate *is_usedvar/1* returns *true* if a state variable *Progress* is used. Superposing this observer onto an EFSM would generate a test suite with *all* symbolic test cases in the EFSM where *Progress* is used. Note that the same observer automata and EFSM with observer usage *observer* would only generate a test suite with the *first* symbolic test case found in the EFSM where *Progress* was used. \square

Typically it is desirable to separate projection and test suite generation. An observer automata used as a filter, and a second observer automata used for specification of a coverage criterion, may be used concurrently. In this case we have a separate symbolic superposition run for each observer automata. Both need to have reached an accepting observer state for the corresponding test case to be included in the test suite.

5.5.2 Model Checking of a Specification

In model-based testing correctness of the specification is crucial. For example, we might want to validate that a stop location can always be reached, or that there always should be a certain ordering of events, size of paths etc. To verify properties in a formal specification there exists a large number of existing model checkers and theorem provers. However, use of any external tool requires translation of an *ERLANG/EFM* specification to an input format supported by that tool. Alternatively, our test suite generation technique requires exploration of the state space (as a model

checker) and the observer automata can be seen to express some property which is false when an accepting observer state is reached, as in the automata-theoretic approach to model checking [Vardi 86]. To validate properties, expressible as abserver automata, the algorithm in Figure 5.2 needs modification, resulting in the algorithm is shown in Figure 5.5. Usage of observer automata as a validation property is declared with an observer usage declaration set to **property**.

```

WAIT := {⟨⟨l0, s0s, true⟩ ∥ {⟨q0, true⟩} ∥ ε⟩}
while WAIT ≠ ∅ do
  select_and_remove ⟨⟨l, σ, G⟩ ∥ Q ∥ ω⟩ from WAIT
  if l ∉ Lstop then
    for all ⟨⟨l, σ, G⟩ ∥ Q ∥ ω⟩  $\xrightarrow{a(\bar{p})/\bar{b}}$  ⟨⟨l', σ', G'⟩ ∥ Q' ∥ ω · a( $\bar{p}$ )/ $\bar{b}$ ⟩
      add ⟨⟨l', σ', G'⟩ ∥ Q' ∥ ω · a( $\bar{p}$ )/ $\bar{b}$ ⟩ to WAIT
    else if ∃ accepting ⟨ι( $\bar{d}$ ), true⟩ in Q then
      return false
  return true

```

Figure 5.5. A symbolic state space exploration algorithm for property validation.

6. Introducing **ERLY MARSH**

In this chapter, we give an overview of **ERLY MARSH** - a tool for model-based test suite generation and test suite execution. Development of **ERLY MARSH** started in 2002 with the aim to implement the techniques introduced in this thesis.

Figure 6.1 shows an overview of the main features of **ERLY MARSH**. In the Figure, ovals represent input specifications that can be processed by **ERLY MARSH**, rectangles represent components of **ERLY MARSH**, and rectangles annotated with “Note” represent outputs from components of **ERLY MARSH** (some of which can also be further processed by other components of **ERLY MARSH**). Thus, the figure shows the components of **ERLY MARSH** that can be involved when processing an **ERLANG/EFSM** specification, together with a specification of coverage criteria represented by an observer expressed in **ERLANG/OBS** to generate test suites and other documents, such as reports on test execution results.

The chapter is structured as follows. Section 6.1 outlines the Prototyper and Simulator, Section 6.2 outlines the Model compiler, Section 6.3 outlines the Pretty printer, Section 6.4 outlines the Test suite generator, Section 6.5 outlines the Test suite execution tool, Section 6.6 outlines the **ERLY MARSH** Verificator, and Section 6.7 outlines the Test suite report tool.

6.1 Prototyper and Simulator

The Prototyper implements the translation of an **ERLANG/EFSM** specification into an **ERLANG** module that can be compiled to executable code. This translation is described in Section 2.11. Utilizing the the executable **ERLANG** module, the **ERLY MARSH** Simulator create a simple simulator environment, in which a user can interact. This environment has an HTML-based graphical user interface, in which a user can set configuration data, and send input events to an **ERLANG** process which executes the generated executable **ERLANG** module. Upon receiving an input event, the **ERLANG** process executes the corresponding transition until it reaches the next state, and sends generated output events back to the GUI via **ERLANG** messages. Additionally, the trace of input and output events is collected and shown in the GUI in the form of a Message Sequence Chart (MSC) [ITU-T 99a].

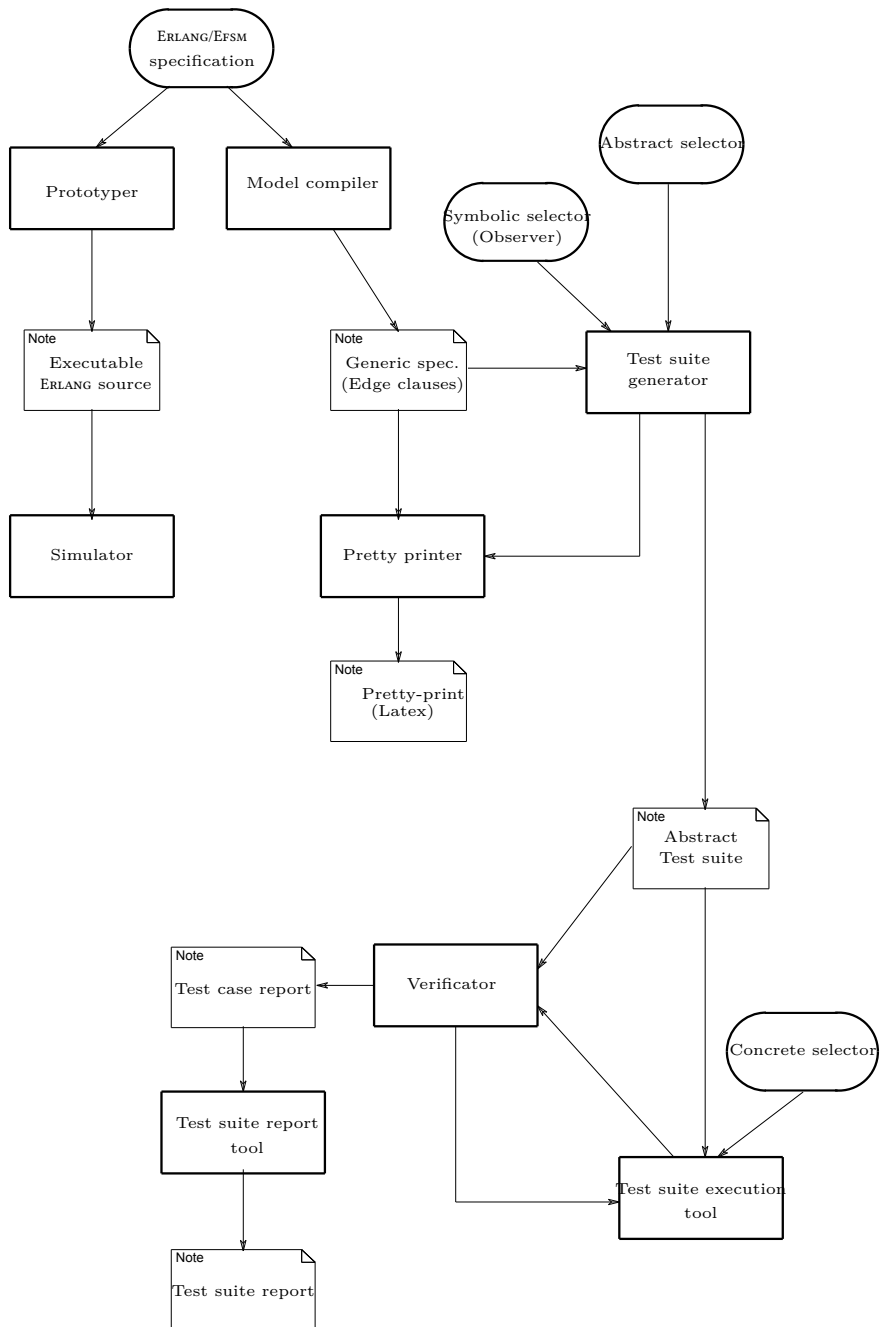


Figure 6.1. Overview of features in the ERLY MARSH tool.

6.2 Model compiler

The **ERLY MARSH** Model compiler is responsible for parsing an **ERLANG/EFSM** specification, given as input, and transforming it into a generic **EFSM** representation, by unfolding transition clauses into edge clauses. The unfolding procedure is described Section 2.10 as part of executing symbolic transitions. After unfolding, a user can choose whether the generated **ERLANG/EFSM** edge clauses should be normalized, as described in Section 2.10. The **ERLY MARSH** Model compiler use an internal representation of **ERLANG/EFSM** edge clauses. This is also the representation of **ERLANG/EFSM** edge clauses assumed by the Test suite generator, see Section 6.4, when generating a test suite.

The Model compiler allows the user to control the transformation in the following ways. The user can:

- choose the maximum recursion depth to be used when handling user defined recursive function in the **ERLANG/EFSM** specification, and
- specify (optional) projection properties on symbolic parameters.

Below, we comment further on these aspects.

Bounding the Recursion Depth

The syntax and operational semantics for **ERLANG/EFSM** user defined functions (as given in Sections 2.3 and 2.4), allows the use of recursion in user defined functions. To illustrate this, consider the following user defined function, which can return a list of arbitrary length (which is at least the value of **FPar**).

```
integer_list(0,Out) ->
    Out;
integer_list(FPar,Out) ->
    integer_list(FPar-1,[Fpar|Out]).
```

When this function is used in a transition clause, such as

```
loc(et,Par) ->
    StateVar=integer_list(Par,[]),
    {next_state,other_loc}.
```

for a location **loc**, with an input event expression **et(Par)**, the state variable **StateVar** is bound to the result of executing **integer_list(Par,[])**. If we assume **Par** is an integer in the domain $\{0, 1, 2\}$ this would cause the Model compiler to create three **ERLANG/EFSM** edge clauses, one for each possible value of **Par**.

```
loc(et,Par) when Par==0 ->
    StateVar=[],
    {next_state,other_loc};
loc(et,Par) when Par==1 ->
```

```

    StateVar=[1],
    {next_state,other_loc};
loc(et,Par) when Par==2 ->
    StateVar=[2,1],
    {next_state,other_loc}.

```

During the symbolic execution performed during test suite generation, we may potentially generate large symbolic expressions with unbound symbolic parameters. The Model compiler limits the size of occurring expressions by imposing a maximum depth at which recursive function calls can be applied. This maximum depth can be given by the user when invoking `ERLY MARSH`. If, during test suite generation, a termination condition is not encountered before this depth is reached, a warning is generated, and test suite generation terminated.

Projection Properties

Sometimes, we may not be interested in considering a complete `ERLANG/EFSM` specification for test suite generation. It may then be convenient to bind symbolic parameters and/or state variables to fixed values when generating a test suite. Although this can be done with observers, it is more efficient to specify such bindings to symbolic parameters already by the Model compiler, as it may limit the state space that needs to be explored. We refer to such static bindings of symbolic parameters and state variables as *projection properties*. By defining a projection property on a global variable v , we ensure that for each step in a symbolic run, v is always either

- not bound to a value, or
- bound to exactly the same value as stated by the projection property.

The use of projection properties is optional.

6.3 Pretty printer

Sometimes (e.g., when debugging the specification), it is desirable to be able to inspect the edge clauses generated by the Model compiler in more detail. `ERLY MARSH` provides a Pretty printer that can generate a \LaTeX representation of the generated edge clauses, in which each edge clause is annotated with the corresponding source lines in the original `ERLANG/EFSM` specification. Furthermore, if a test suite has been previously generated, coverage items can be visualized in the generated \LaTeX document. As each coverage item corresponds to an accepting observer location with observer parameters that (typical) originate from the `ERLANG/EFSM` specification, each observer parameter is assigned a unique color. This color is then used to color the corresponding syntactical expression in the generated \LaTeX document. For example, edge coverage

can be used to identify “dead code” in the `ERLANG/EFsm` specification by assigning a unique color to all covered edges in the generated `LATEX` document.

6.4 Test suite generator

The `ERLY MARSH` Test suite generator is responsible for generating abstract test suites from the generic `EFsm` representation of the `ERLANG/EFsm` specification, symbolic selector (observer expressed in `ERLANG/OBS`), and an abstract selector using the techniques described in Section 5. Both symbolic and abstract test suites are generated off-line.

The intention of the Test suite generator is to provide a large degree of flexibility when generating a test suite, by giving the developer control of various aspects of test suite generation. The techniques presented in this thesis provide several different ways to exercise such a control.

- The generation of test suites can be controlled using observers, e.g., by specifying coverage criteria.
- The size of the generated symbolic test suite can be limited to only include a maximum number of coverage items, or a maximum number of selected symbolic test cases.
- The generation of abstract test cases from symbolic test cases (Abstract selector in Figure 6.1) can be performed by using either the **Biased** or **Rndsym** abstract test case selection technique, as described Section 5.4.1. For symbolic test suites of modest size, it is also possible to select *all* possible abstract test cases.
- The selection of concrete test cases from abstract test cases is handled by a SUT-dependent call-back module implemented in `ERLANG`, and used by the Test suite execution tool, described in Section 6.5.
- The developer can choose between several different internal representations of path and superposition conditions in symbolic traces. These representations are described in more detail below.

In addition, the `ERLY MARSH` Test suite generator implements a number of optimizations. For example, all edge clauses are mapped to a unique integer so that they can be easily distinguished by, e.g., the observer predicate `is_edge/1`.

Representing generated symbolic test suites

Recall from Section 5.1 that a symbolic test case consists of a symbolic trace with input and output expressions, and a conjunction formed from a path condition and a superposition condition. In `ERLY MARSH`, the symbolic trace is represented as a list with `ERLANG` records, encoded into a more compact representation when stored off-line. For the conjunction of path and superposition condition the most efficient representation is

specification dependent. For example, if symbolic parameters and state variables are known to range over small finite domains there exists several efficient representations. A lesson learned from the Evaluation in Section 7 is that the choice of representation of path and superposition conditions is important for efficient test suite generation.

Let us here provide more details on the supported internal representations of path and superposition conditions. **ERLY MARSH** currently supports three different representations.

- *Binary Decision Diagrams* (BDDs) [Bryant 86], which can be used when parameters and variables range over boolean domains.
- *Numerical Decision Diagrams* (NDDs) [Asarin 97], which can be used when parameters and variables range over small finite integer domains. In an NDD representation, a parameter or variable which ranges over a domain of size K can be encoded using $\lceil \log K \rceil$ bits. Thus, a condition involving d variables, each ranging over a domain of size K , can be represented by a boolean function of $d * \lceil \log K \rceil$ boolean variables. When using NDDs the domain of each parameter and variable must be known *a priori*. Therefore, it is essential that the domains for all symbolic parameters and variables to be represented by an NDD are known, e.g., from provided type declarations. In **ERLY MARSH** the default domain for variables and functions, that are not given any explicit type declaration, is boolean.
- **ERLANG** guards, as occurring in **ERLANG** clauses. These are symbolic **ERLANG** expressions, where parameters and variables can range over any domain that is supported in **ERLANG**. This allow us to express path conditions where we are not limited to symbolic parameters with boolean (BDD) or small integer (NDD) domains.

After generating a test suite, the Test suite generator may feed the Pretty printer (described in Section 6.3) with additional information on coverage items covered in the generated symbolic test suite.

6.5 Test suite execution tool

The **ERLY MARSH** Test suite execution tool is responsible for

- initializing a test execution environment (e.g., start protocol adaptors) so that a concretization of each abstract test case can be executed,
- translating between abstract and concrete test cases; this translation uses an on-the-fly translation implemented by an **ERLANG** call-back module, as further outlined in Section 5.4.2, and
- on each event received from the SUT by the test execution environment, passing the (abstracted) event to the **ERLY MARSH** Verificator for verification of correctness.

Let us comment on some aspects of the test execution environment. Execution of a generated test suite requires that we are able to set up a test execution environment that interacts with the SUT as described by each abstract test case. Each test case typically represents some particular combination of data settings that determine, e.g., values of parameters in input events. If the test execution environment should be able to handle concurrent requests, we need a mechanism to distinguish these data settings. For this purpose, **ERLY MARSH** uses a *test case identifier* to identify the data settings in an actual test case. The data settings prescribe abstract values of input expression parameters and configuration parameters, but may also be used to control responses from the environment, which in general can be non-deterministic. The test case identifier is then encoded into a suitable parameter in each concrete event sent to the environment where it is decoded by the abstraction translation so that the abstract test case currently executed can be uniquely identified. Likewise, in any concrete event sent from the environment the concretisation translation encodes the test case identifier. Note that

- parameters holding the test case identifier in the concrete events may not have mappings to parameters visible in the specification used to generate test suites, and
- the SUT must use these concrete parameters (e.g., a sequence numbers) internally to distinguish between different test cases.

See also Section 7.1.3 for an explanation on how test case identifiers were used in the Evaluation.

Example 6.1 Assume we want to execute a test suite generated from the **ERLANG/EFM** specification in Example 2.1. We must then identify concrete parameters in the implementation to identify test cases. For example, for *checkin*(*Day*) and *checkout*(*Day*) events one can assume the existence of some revision information (e.g., date) on files checked in and checked out, available in a concrete protocol, that could be used. Similar, for the *wakeup*(*Day*), *progress*(*X*) and *incident*(*I*) events one can assume some timestamp including e.g., the date. An actual implementation needs to handle the actual encoding of test case identification in the call-back module, when translating between abstract and concrete events. Note that to be able to execute test cases concurrently, the implementation of the configuration access function *daytype*(*Day*) needs to be parameterized by a parameter used as test case identifier. \square

The Test suite execution tool supports the execution of test cases both sequentially, or concurrently. Execution is handled by distributing execution to a given number of **ERLANG** processes, possibly located on different hosts. Whenever ready, each such (handler) process requests a new test case to execute from a single manager process, holding the test suite,

Note that there may exist limitations on the concrete events (e.g., window size in an underlying protocol, or domain of parameter) or in the SUT (e.g., limited possibility to handle simultaneous configurations) limiting the number of possible concurrently executing test cases.

6.6 ERLY MARSH Verificator

The **ERLY MARSH** Verificator verifies executed test cases for correctness against corresponding test cases in the generated test suite. On test execution, each event received by the test execution environment is verified against the abstract test suite. Additionally, the **ERLY MARSH** Verificator may also use a SUT specific call-back module to verify SUT specific data. For example,

- text logs may be searched for strings that may be suspected to indicate an error, e.g., the string “error”,
- counter values may be compared before and after test case execution for suspected updates, and
- resource utilization data may be watched for abnormal usage in the SUT

If not found correct, the **ERLY MARSH** Verificator aborts test case execution.

Sometimes it is also desirable to control what is verified as successful execution. E.g., see Section 8.1.4 for an example where verification of the output parameters are limited. Verification control is handled by configuration. The **ERLY MARSH** Verificator returns a *Test case report* on each test case verified. This report may be fed into the **ERLY MARSH** Test suite report tool.

6.7 Test suite report tool

After executing a test suite, the **ERLY MARSH** Test suite report tool summarizes the types of failures found and (possibly) generates a report on each single test case executed.

Executing large test suites may cause a large number of failing test cases, many of which may detect the same fault. In particular for large test suites, with many failing test cases, it is therefore essential that the results are organized in such a way that a user can easily understand the outcome of executing the test suite. The **ERLY MARSH** Test suite report tool uses *concept analysis* [Wille 82, Ammons 03] to automatically find similarities between failing test cases and *cluster* them together. Concept analysis clusters failing test cases into a hierarchy of small clusters within big clusters, where failing test cases in a small cluster are more similar

than failing test cases in a big cluster. Thus, a user may inspect a small set of clusters of failing test cases, instead of looking at a much larger set of individual test cases. For each cluster, a summary is provided which typically includes a symbolic trace and information on the detected failure, see Figure 6.2. The idea is that it should be possible to fast get a rough understanding of detected failures and possible faults causing the failures.

For each executed test case a report can be generated that includes values used for instantiated symbolic parameters, part of the trace executed, details on why **ERLY MARSH Verificator** finds the the execution to fail (if any), a Message Sequence Chart representation of the executed test case, and possibly additional data (such as logs, counters and resource utilization) from the SUT. It can be noted that a slightly enhanced Message Sequence Chart, including names for entities with which the SUT communicates, can be generated if input and output events are declared to return a record `#em_i{}`. For example, a `psi/2` event may be declared by

```
-spec psi(VLRid::any(),IMSI::any()) ->
    #em_i{dir::out, env\_node::'MSC', env\_par::'VLRid'}).
```

in the specification, if used as an output event for communication from the SUT to an MSC (Mobile Switching Center). In Figure 6.2 it can additionally be noted that there exists multiple MSC entities. Which MSC a particular `psi/2` event is sent to is controlled by a parameter in the event (`VLRid` in this case).

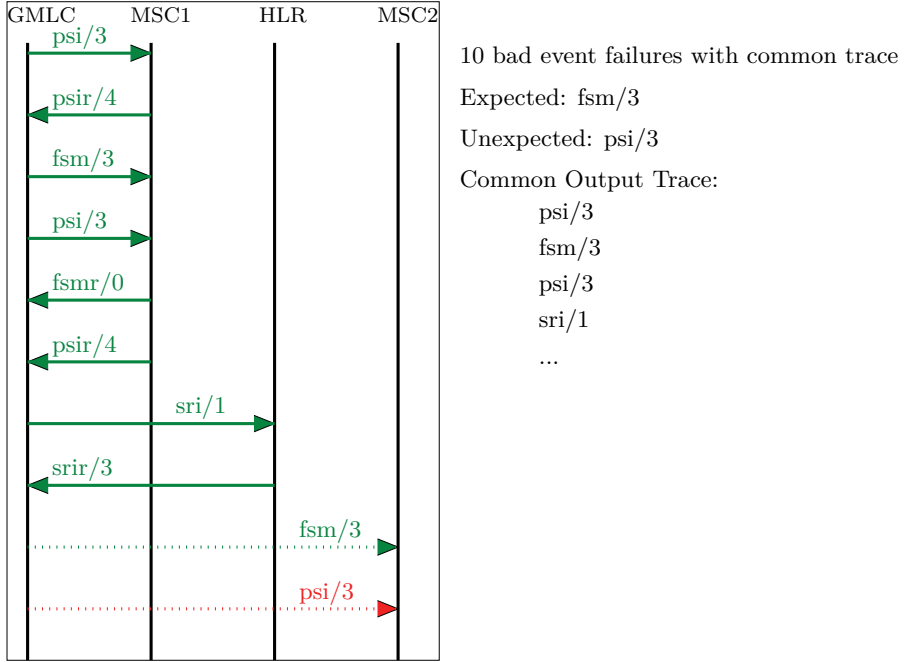


Figure 6.2. A screen dump from ERLY MARSH after executing a test suite with failing test cases where GMLC is the SUT and MSC1, MSC2 and HLR are parts of the test execution environment. The figure shows how 10 test cases form a cluster by sharing a trace (green solid lines), ending with a received *srir/3* input event. After this event, all test cases in the cluster expected the SUT to send an output event with event type *fsm/3* to MSC2 (green dotted line), but unexpectedly the SUT instead sent an output event with event type *psi/3* to MSC2 (red dotted line). This particular cluster was generated from the A-MLC (Advanced Mobile Location Center) specification elaborated further in Section 7.1.1.

7. Evaluation: Testing a Telecom Software Application

In this chapter, we present the setup for our evaluation of how the techniques for model-based test suite generation, presented in this thesis, perform when applied to existing industrial protocols. Thus, we seek to evaluate if model-based testing in general, and the **ERLY MARSH** tool in particular, is feasible in a real industrial environment and how it can be used most efficiently. We do this by creating a formal specification for an existing industrial protocol, from which test cases are automatically selected, executed and validated. We then evaluate selected techniques based on coverage based testing and random testing, and compare the results with an existing “manually” created test suite. In particular we will study different strategies for:

Selecting test cases for a test suite, to test as efficiently as possible (low cost) while gaining maximum coverage and fault detection (benefits). We consider both the selection of symbolic test cases, and the selection of abstract test cases.

Executing the set of selected test cases as efficiently as possible with respect to time and fault detection effectiveness.

Further we discuss the cost of creating the required **ERLANG/EFM** specification. The evaluation has the form of a case study, in which we generate and execute several test suites for testing a commercially available telecom software system, **A-MLC**. The version of **A-MLC** used in the case study was not seeded with any faults. All faults found are “real” faults, not (yet) detected by any other kind of testing. This means that our evaluation measures the capability of detecting faults that typically occur in industrial settings, in contrast to studies that are based on faults generated by seeding or mutation.

A purpose of our study is to evaluate different techniques for generating test suites. Our hypothesis is that test suites may be of a measurable quality, and that the quality can be controlled by techniques for generating test suites. We therefore organized the evaluation by defining variables we can control, called *independent variables*, i.e., parameters that control the generation and execution of test suites, and the variables that measure the quality of a test suite, called *dependent variables*, i.e., observable effects of the different values of the independent variables. When choosing selection techniques we had two main goals: (1) to compare different

techniques for generating test suites, including manual, random and coverage based techniques, and (2) to compare some well-known coverage criteria discussed in more detail in Section 4.

The chapter is organized as follows. Section 7.1 presents the setup of the case study outlining the SUT, the ERLANG/EFSM specification of the SUT, and the test execution environment, Section 7.2 explains the independent variables, Section 7.3 explains the dependent variables, and Section 7.4 discusses threats to validity of the case study.

7.1 Mobile Arts Advanced Mobile Location Center

Mobile Arts A-MLC (Advanced Mobile Location Center) allows Mobile Network Operators to provide presence data about subscribers with a mobile device to presence dependent applications. Essentially it acts as a standard Gateway Mobile Location Center (GMLC) node [3gp 00], with the ability to provide additional presence data not required by a standard GMLC. The supported presence data includes details about the location of mobile devices, as well as about their current status and capabilities. For example, a taxi switchboard application may want to know where a calling user is located, in order to send the closest available taxi car to the customer. A-MLC is commercially available and has been deployed with several mobile network operators within Europe and Asia.

Figure 7.1 illustrates how A-MLC interacts with other entities in a telecom network system. A presence dependent application communicates with A-MLC using Mobile Location Protocol (MLP) [oma 04], a standard XML based protocol utilizing HTTP over IP. To provide presence data for a mobile device, A-MLC uses the GSM/UMTS core network, from which the information is retrieved. In this network, a few nodes are of particular importance, including:

- HLR (Home Location Register), a central database that contains details of each mobile device with a subscription at an operator,
- MSCs (Mobile Switching Centers), each of which is responsible for routing voice calls and SMS in a certain area,
- VLR (Visitor Location Register), a database of the subscribers who have roamed into the area of the MSC which it serves, and
- BSC (Base Station Controller) handling allocation of radio channels.

For communication with these GSM/UMTS core network nodes, A-MLC uses MAP (Mobile Application Part) [3gp 99], a protocol in the SS7 protocol stack. The presence data we are interested in are stored in the HLR and VLR, but may not always be updated. Thus, typically an MLP request from a presence dependent application is followed by a sequence of MAP requests to (optionally) force an update of cached presence in-

in the specification. Errors in the specification were sometimes caused by the detailed functional specification containing unclear, wrong, or missing information, e.g., on how to decide whether a user is roaming when we have multiple conflicting MAP input events, or how to map a trace of MAP events into an MLP response code. More frequently, however, errors were caused by mistakes (bugs) made by the author when formalizing the functional specification in the `ERLANG/EFsm` language.

The version of A-MLC used for our evaluation was a snapshot made during development of some new functionality, compared to an earlier deployed versions.

7.1.1 The A-MLC `ERLANG/EFsm` specification

The A-MLC `ERLANG/EFsm` specification was created with the goal to capture all of the functionality provided by the detailed functional specification, in a consistent manner. Several methods exist to achieve the goal to retrieve presence data for a subscriber with a mobile device. In Figure 7.2, a graphical outline of the ATI (Any Time Interrogation) presence method is given. Using this method, A-MLC sends an ATI to the HLR that may hold cached presence data on the mobile device. It may happen the cached data the HLR holds is invalid, e.g., because the subscriber is roaming or presence data is too old. In such cases a solution is to force an update of the HLR with an FSM (Forward Short Message), but that requires an MSC address that can be obtained by sending an SRI (Send Routing Info) and wait for the response. The A-MLC `ERLANG/EFsm` specification additionally supports the PSI (Provide Subscriber Info) presence method that shares many similarities with the ATI presence method.

The specification accepts five forms of input event types:

- slir/6* – representing a SLIR (Standard Location Immediate Request), on MLP, event from a presence application,
- atir/4* – representing an ATI response from an HLR (Home Location Register) holding presence data,
- srir/3* – representing a SRI response from an HLR,
- fsmr/0* – representing a FSM response from an MSC with status of the mobile device, containing an address to the MSC serving the mobile device, and
- fsmTimerr/0* – representing an internal timeout when a response was not received within a given time.

The output event types are of form:

- slia/5* – representing a SLIA (Standard Location Immediate Answer), on MLP, event to a presence application,
- ati/1* – representing an ATI request to an HLR,
- sri/1* – representing a SRI request to an HLR,

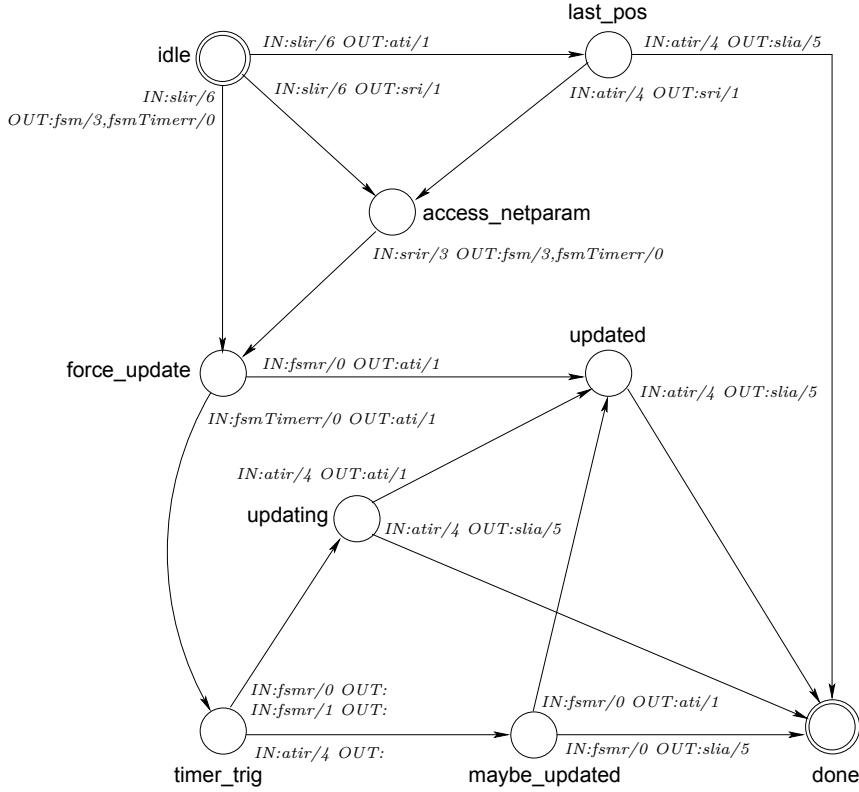


Figure 7.2. A graphical representation of a the part of the state machine created for A-MLC, covering the ATI presence method. Note that due to space restrictions, all lists of event parameters are simplified to a single parameter, which merely indicates whether the real parameters are MAP or MLP parameters. Further, some edges are omitted, and all state variable updates and guards on edges are not shown.

fsm/3 – representing a FSM request to an MSC, and
fsmTimer/0 – representing start of a timer.

There are 9 locations in the specification, corresponding to the circles in Figure 7.2.

idle is the initial location, representing that A-MLC is ready for a new *slir/6* event,

access_netparam represents that we have received the address to the MSC,

force_update represents that we have sent an FSM to force an update of presence data for the mobile device

updated represents that presence data for the mobile device was successfully updated in the GSM/UMTS core network,

timer_trig represents that an internal timer triggered because the FSM response took too long and we therefore try to send the *ati/1* request anyway,

updating represents that the *fsmr/0* response was received but not the *atir/4* response,

maybe_updated represents that the *ati/1* response was received but not the *fsm/3* response, and finally

last_pos represents the case where we are only interested in the presence data currently held in the HLR.

Example 7.1 A typical scenario is found in Figure 7.3. The scenario starts by a *slir/6* request from Application to retrieve presence data for a subscriber with a mobile device. Triggered by the *slir/6* request, the A-MLC sends a *ati/1* event to the HLR, containing a query for presence data for a subscriber. Presence data includes e.g., cell identity to which the cell the subscriber is currently associated, and status of the mobile device (e.g., if busy in a phone call or idling). Since the presence data known by the HLR is too old, A-MLC must force an update. It does this by querying the HLR with an *sri/1* event for the last MSC the subscriber was associated to. After this information is received from the HLR in an *srir/3* event, A-MLC sends an *fsm/3* event to the MSC (i.e., eventually this will cause an SMS, Short Message Service, to be sent to the subscriber) forcing an update of the current presence data in the HLR. Now it happens that the response to the *FSM*, from the MSC, takes too long, causing a timeout event *fsmTimerr/0* occurs. A-MLC then sends a new *ati/1* event to the HLR, hoping that the presence data has already been updated. Eventually, both *fsmr/0* and *atir/4* events appears, but the presence data received from the HLR is still not updated. The reason for this might be that the *ati/1* event was sent too early and we send a final *ati/1* event to the HLR. After the last *atir/4* is received, A-MLC returns presence data back to Application in the *slia/5* event. □

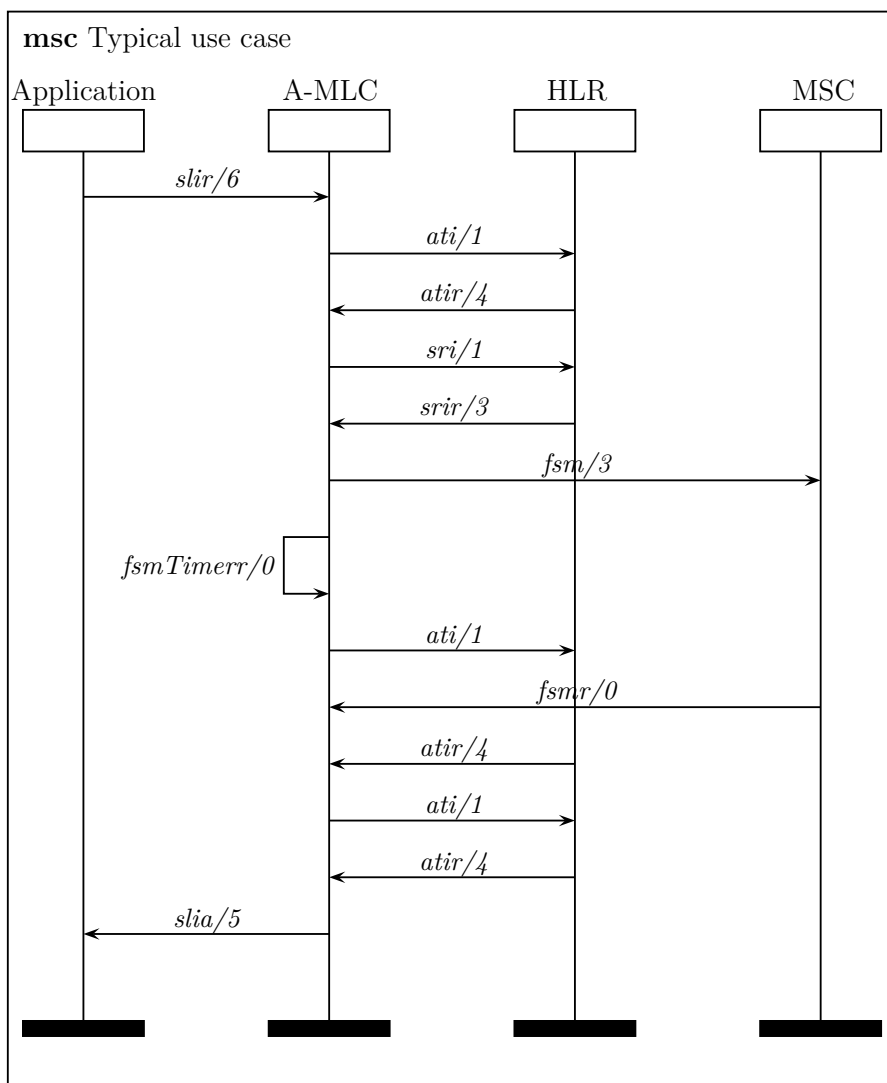


Figure 7.3. Typical execution of a test case in which execution starts by sending a *slir/6* request to the A-MLC, the A-MLC then executes a number of SS7 MAP operations (i.e., *ati/1*, *sri/1* and *fsm/3*) and returns a *slia/5* response. Note that due to space restrictions, all event parameter lists are simplified to a single parameter only indicating if real parameters are MAP or MLP parameters.

Due to the generic nature of the involved protocols that communicate with the environment (MLP and MAP), these interfaces were only partially modeled in the A-MLC `ERLANG/EFSM`. That is, we only model *well-formed* events that can occur on these protocols. Events found to be malformed, and thus rejected on these interfaces, were not modeled. Further, for each protocol, we only model the subset of well-formed events considered important to generate test suites from. The resulting A-MLC `ERLANG/EFSM` specification captures all possible traffic sequences through A-MLC via the MLP protocol towards a presence dependent application, and all relevant MAP operations towards the GSM/UMTS core network. Lower level protocols in the IP stack (e.g., TCP) and SS7 stack (e.g., TCAP) are not part of the formal specification. Likewise, no Operation and Maintenance interface (counters, alarms, GUI etc.) are part of the formal specification. Furthermore, with the additional knowledge that the A-MLC implementation makes use of `ERLANG`'s light-weight threads to separate requests, the handling of concurrent requests was not considered to need further verification and was therefore omitted from the A-MLC `ERLANG/EFSM`. In Figure 7.2 only the ATI presence method is included. The complete specification also includes support for the PSI presence method, introducing additional events and locations. Selection of which presence method to use is handled by configuration data. In total, the specification used in the evaluation consists of 12 locations, 11 input event types, 8 output event types, 70 input event parameters, and 15 configuration parameters. Typically the configuration parameters were reflecting network capabilities. For example, what presence method to use, and if there exists required agreements when retrieving presence data in networks not controlled by the subscribers operator.

The A-MLC `ERLANG/EFSM` specification does not handle concurrency. As we do not validate correct concurrent behavior, we rely on some knowledge about the implementation of the SUT. First, the SUT is implemented in `ERLANG` with native support for concurrency by the use of `ERLANG` processes. Second, the SUT handles concurrent requests by a separate `ERLANG` process for each request. Thus, assuming `ERLANG` implements concurrency correctly the benefit of validating correct handling of concurrency is limited. In general, for implementations in `ERLANG`, there is a limited need to specify concurrent behavior of requests, if we can rely on `ERLANG` to handle concurrency correctly.

7.1.2 Symbolic test suite generation

The created specification allowed us to generate large numbers of different symbolic test cases. For the evaluation, the largest test suites even became too large for making the experiment manageable with available

hardware. We therefore additionally used a projection property, as describe in Section 6.2, to project the specification to a model with less functionality. Thereby, we did not cover use cases including the use of an internal cache in A-MLC, and handling of the sub-result received when forcing an update of cached presence information in VLR and HLR.

Depending on the selection technique, test suite generation also may be expected to depend on the structure of the specification. It was therefore decided that the evaluation should include also a comparison with a normalized generic specification (see Section 2.10), in which the number of edge clauses had been reduced by merging similar edge clauses. Since some parts of the implemented SUT was known to be poorly tested (because of ongoing development), we also decided to perform an evaluation on the more “well-tested” part of the SUT.

Each symbolic test case in the symbolic test suite must be represented in a way that allows efficient creation and manipulation of path and superposition conditions. Since many parameters of the A-MLC `ERLANG/EFSM` specification range over small finite domains, we have chosen to use NDDs (*Numerical Decision Diagrams*), as described in Section 6.4. As the small finite domains in the A-MLC `ERLANG/EFSM` specification are not necessarily integer domains, mappings between each used value and an integer were created.

7.1.3 Test Execution Environment

To be able to execute test cases we need, apart from a SUT (A-MLC), a Test Execution Environment. See Figure 7.4 for an overview. We used a setup with 3 `ERLANG` nodes;

- **ERLY MARSH Node** holds the part of the Test suite execution tool, selecting and initiating test cases to be executed, and all handling of configuration data. The Test suite report tool generates a report when execution has finished.
- **Environment Node** holds the part of the Test suite execution tool, creating abstractions of incoming events and concretizations of outgoing events. Abstractions of incoming events are verified by the **ERLY MARSH Verificator**. To create and execute the concrete events against the SUT, dedicated protocol layers (i.e., for MLP and MAP) were used. For the MAP layer we had two options when executing. The “Simulated” MAP option utilized an `ERLANG` remote procedure call and made it possible to run both the **Environment Node** and **A-MLC Node** on the same host, but also shortcut execution of all SS7 operations (including MAP). The “Real” MAP option utilized dedicated software and hardware for SS7 stacks.

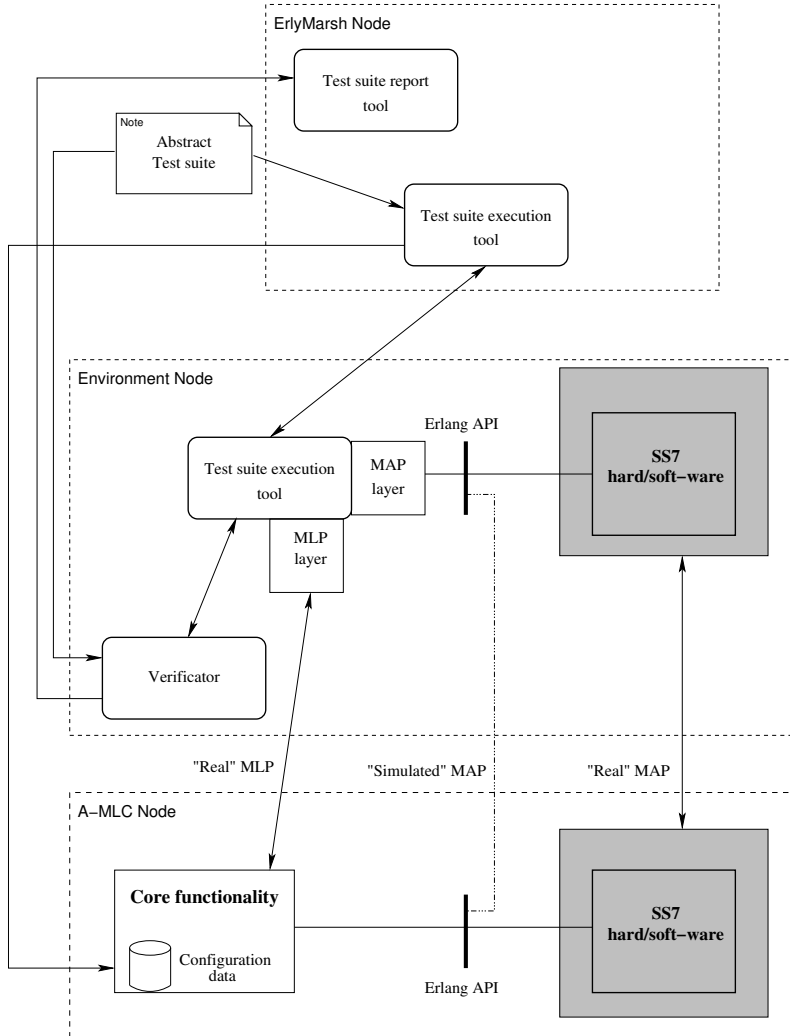


Figure 7.4. Overview of the test execution environment for the A-MLC product. The “grey” parts were dedicated SS7 hard/soft-ware not used in this evaluation.

- **A-MLC Node** holds the SUT. The dedicated software and hardware for SS7 is a mandatory part when deployed for execution in a real network. For test execution, it was possible to execute the Core functionality, i.e., the part of the SUT on which A-MLC **ERLANG/EFSM** specification focused, without the need for software and hardware for SS7.

Hardware used for all test suite generation and test suite execution was an Asus V1S laptop (Intel Core 2 Duo T7300 2 GHz and 2 GB RAM).

Since the test execution environment is distributed, we must assign a unique test case identifier to each test case, which is encoded into some parameter of messages, as described in Section 6.5, For A-MLC, the parameters that are used for this purpose include

- in events on the MLP interface, the parameter MSISDN (Mobile Subscriber ISDN), representing the subscriber that should be positioned, and
- in events on the MAP interface, either MSISDN, when available, otherwise IMSI (International Mobile Subscriber Identity).

Note that the parameter holding the test case identifier may have different domains on different interfaces. Thus, when utilizing different interfaces, if different values are used they must be kept synchronized.

Before execution of a test case, abstract values are assigned to configuration data. After concretization A-MLC is then configured with the concrete configuration data. Next, the abstract test case is stored in the Test suite execution tool and **ERLY MARSH Verificator**. During execution of a test case, each concrete event sent from the SUT to the Test suite execution tool is translated into an abstract event, and sent to the **ERLY MARSH Verificator**. If the **ERLY MARSH Verificator** successfully verifies the abstract event, the Test suite execution tool continues with the next event in the test case. For example, if the next event is an input event to the SUT, it is made concrete and sent to the SUT utilizing a protocol layer. If the **ERLY MARSH Verificator** is not successful verifying, all pending requests are aborted and execution ended. In this case study, concretizations and abstractions are translations between input/output events sent to/from the SUT. In **ERLY MARSH** translations are handled by two configurable tables, one for abstractions and one for concretizations. After the test case has finished executing, the results are collected and all relevant data stored in a database. Later, after executing the test suite, this data is used to present the results to the user via the HTML based user interface.

Figure 7.5 shows how the test case from Example 7.1 is executed in the test execution environment. The test case begins by configuring A-MLC with the configuration data, and providing both the Test suite execution tool and **ERLY MARSH Verificator** with the abstract test case to be executed. Execution of the test case is initiated by sending a *slir/6* request

to the SUT. The SUT then responds by executing a number of SS7 MAP operations before it returns a *slia/5* response. After the test case has finished, the configuration needs to be reset and results from the **ERLY MARSH** Verifier are reported back to the Test suite report tool at the **ERLY MARSH** Node.

Test cases can be executed sequentially or concurrently. Due to the use of configuration data in the A-MLC **ERLANG/EFMS**, concurrent execution of test cases requires A-MLC to handle many concurrent configuration data sets. Fortunately this is supported in A-MLC. Since handling of concurrent requests was left outside of the A-MLC **ERLANG/EFMS** specification, the choice whether execute sequentially or concurrently could be made without affecting the functionality of the SUT. A drawback of concurrent execution is that potential resource failures, such as checking that used memory is returned properly, can not be traced to individual test cases. Our testing revealed in total one such possible resource failure. Also, when executing a test case A-MLC may generate logs that may reveal additional (hidden) failures, not discovered otherwise. Using concurrent execution, **ERLY MARSH** did not support distinguishing logs from different test cases. As can be expected, a significant speedup was achieved by running test cases concurrently.

7.2 Independent variables

The *independent* variables are the variables in the evaluation that are controlled. Here, the identified independent variables mainly concern the selection and execution of test cases. We are interested in studying the effect of different values of these variables.

Recall that test cases can be represented on three abstraction levels: the symbolic, the abstract, and the concrete level. Using the observer automata, introduced in Section 3.1, test cases are selected on the symbolic level. Each selected symbolic test case represents a set of abstract test cases. An abstract test case can be selected, either from a given symbolic test case or by some other means e.g., random or manual. Thus, another independent variable is the selection of abstract test cases. From each abstract test case a concrete test case needs to be selected. In this case study we do not study the selection of concrete test cases but rely on a well defined one-to-one mapping between each abstract and concrete value. Whenever a concrete test suite is selected we also need to select how the test suite should be executed on the SUT. In Section 7.2.3 we consider executing complete test suites sequentially or concurrently.

Thus, the four classes of the independent variables are (1) Symbolic test case selection techniques, (2) Abstract test case selection techniques, (3) Test execution strategies, and (4) Base model from which symbolic test

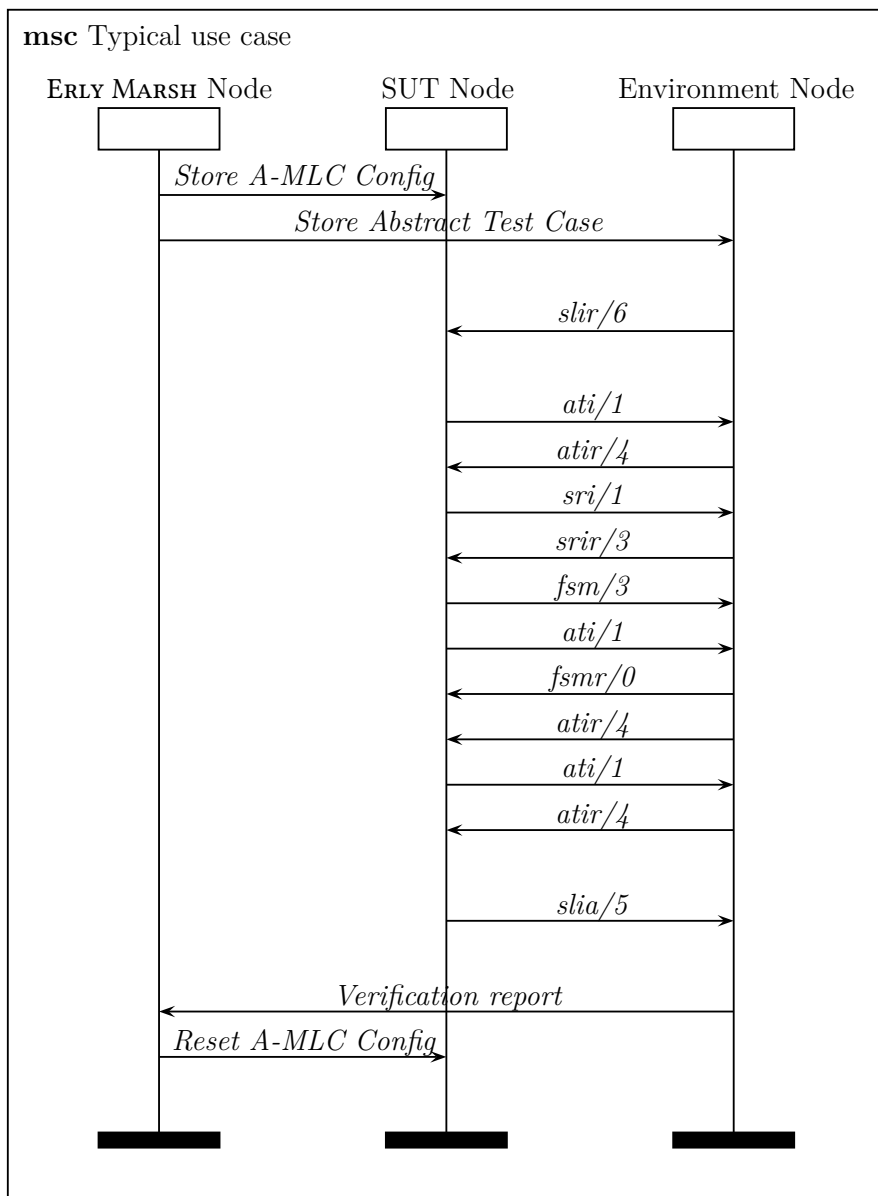


Figure 7.5. Typical execution of a test case in a simulated environment where execution start by configure A-MLC with configuration data and the abstract test case in the Test suite execution tool and ERLY MARSH Verificator in the Environment Node. The test case is then initiated by sending a *slir/6* request to the A-MLC, the A-MLC then executes a number of SS7 MAP operations (i.e., *ati/1*, *sri/1* and *fsm/3* with corresponding responses) and returns a *slia/5* response. After the test case has finished, the configuration needs to be reset and results from the ERLY MARSH Verificator are reported back to the Test suite report tool at the ERLY MARSH Node. Note that due to space restrictions, all event parameter lists are simplified to a single parameter only indicating if real parameters are MAP or MLP parameters.

cases are selected. In the following sections we discuss these independent variables in more detail.

7.2.1 Symbolic test case selection techniques

We considered four different symbolic, coverage based, test case selections (coverage criteria) for the A-MLC `ERLANG/EFsm`.

These were:

- *All paths* (All-Paths) cover all possible paths by making an exhaustive unfolding of all paths in the base model,
- *All target locations* (All-Locs) cover all possible target locations at least once,
- *All edges* (All-Edges) cover all possible edges at least once, and
- *Definition-Use pair* (Def-Use) of state variables cover all possible pairs of definition and (at least one) usage of state variables.

See Section 4 for more detailed info, including observers, for these coverage criteria. Note that, although it is possible to use the observer automata to generate All-Paths test suites, we did instead use `ERLY MARSH` to simply search for all possible traces with a start location and a stop location.

7.2.2 Abstract test case selection techniques

A symbolic test case represents a set of abstract test cases. By assigning values to all symbolic parameters, i.e., the input expression parameters and configuration data, an abstract test case can be obtained. The abstract test case selection techniques determine how to select a few of many possible abstract test cases obtained from a single symbolic test case. The abstract test case selection techniques considered in this evaluation are **Biased** and **Rndsym** abstract test case selection, see Section 5.4.1. We here assume the formal specification to be detailed enough to only make it necessary to select a single abstract test case from each symbolic test case.

Random abstract test case selection

Randomly generated test suites were generated by first generating the symbolic test suite with all symbolic test cases using the original base model (All-Paths^{org}), and then selecting random abstract test cases in two steps: (1) Randomly select a symbolic test case with some probability and (2) from the selected symbolic test case, randomly chose an abstract test case from those covered by the selected symbolic test case. In order to make this selection correspond to a selection in actual random test generation, we must assign a *weight* to each symbolic test case, which is

proportional to the probability that a randomly generated abstract test case is an instance of this symbolic test case. We have identified two different ways to assign weights to symbolic test cases, corresponding to different views on how test case selection is performed in random testing.

Test case weight We assume the probability of each possible abstract test case to be equal. The weight (W^{tcw}) of a symbolic test case is then the number of possible different abstract test cases that can be instantiated from it.

Event weight We consider that random testing is a process, in which each consecutive input event is selected randomly among the ones that are expected by the ERLANG/EFSM model in its current state. This view seems more in accordance with actual random testing, which sends a sequence of input events, each randomly selected, to the SUT. The weight of a symbolic test case will then be proportional to the product of the probabilities that each input event is included in the symbolic test case. We identify two different ways to interpret this view, resulting in the two different ways to assign weights: *simple event weights (sew)* and *event weights (ew)*.

The case of *simple event weights* considers a scenario where the selection of an input event is done without knowledge of which events can actually be received by the SUT. In other words, an input is selected without considering which are the preconditions of possible transitions from the current state. In general, a large fraction of selected input events, that can not be received, will then be rejected, causing the corresponding test cases to be aborted. The simple event weights are then the relative weights of the successful non-aborted test cases. To calculate the simple event weight of a symbolic test case, we use the formula

$$W^{sew} = W^{tcw} \times P \times T$$

where

- W^{sew} is the simple event weight.
- W^{tcw} is the test case weight.
- P is the probability for selecting the sequence of event types in a symbolic trace. For example, P^{sew} becomes $\frac{1}{10} \times \frac{1}{10} \times \frac{1}{10} = 10^{-3}$ if there exists 10 possible types of events in the specification, for a symbolic trace with 3 input expressions.
- T is the probability to select one of the abstract test cases represented by the symbolic test case, without considering any constraints given by guard expressions, i.e., without considering whether the SUT can actually receive the selected input event. For example, T becomes $\frac{1}{10} \times \frac{1}{10} \times \frac{1}{10} = 10^{-3}$ if a test case has 3 symbolic parameters, all with a domain of size 10,

The other approach, i.e., the *event weighted* one, considers a random selection only among input data that can actually be received by the SUT (according to the model), i.e., input data that satisfies the precondition of some enabled edge. To calculate the event weight of a symbolic test case, we use the formula

$$W^{ew} = W^{tcw} \times P \times T$$

which is similar to that for simple event weights, but where

- W^{ew} is the event weight.
- P is the probability for *enabled* input event types in a symbolic trace to be selected. For example, P^{ew} becomes $\frac{1}{2} \times \frac{1}{5} \times \frac{1}{10} = 10^{-2}$ if there exist 2 enabled event types in the first location, 5 enabled event types in the second location and 10 enabled event types in the third location, for a symbolic trace with 3 input expressions.
- T is the probability for *enabled* input parameter instances in a symbolic trace to be selected. Letting T_k^{tcw} be the number of possible different abstract executions corresponding to the k first steps of the symbolic test case, and letting T_k^{out} be sum of all enabled computation steps with the input event type in the $(k + 1)$ st computation step, this probability can be calculated as

$$T = \frac{1}{T_0^{out}} \times \frac{T_1^{tcw}}{T_1^{out}} \times \dots \times \frac{T_{n-1}^{tcw}}{T_{n-1}^{out}}$$

for a test case of length n .

In our experience, a longer test case often tends to include a larger number of symbolic parameters not occurring in the path condition. This because it is often the case that the involved input expressions do not have constraints on all parameters. This causes test case weighting to favour longer test cases. Event weighting on the other hand, favours short test cases, since both T and P become smaller for longer test cases.

Note that when calculating the number of possible different abstract test cases that can be instantiated from a symbolic test case, we include *all* symbolic parameters in the symbolic test case, also those not occurring in the path condition.

Manual abstract test case selection

Manually created test suites (hand-crafted) is the traditional way of testing, and the way A-MLC was tested before using any model-based test tool assistance. We define *manual testing* here as the activity of

- generating a list of concrete test cases from a functional specification given as informal textual descriptions and (possibly) message sequence charts, and

- executing a list of concrete test cases, using some test environment and (possibly) executed by some script.

The manual test cases used in this case study were designed to cover 11 “traffic scenarios” defined in the detailed functional specification where each traffic scenario describes a set of similar use cases specified by a hierarchical Message Sequence Diagram and informal text descriptions. Further, the A-MLC **ERLANG/EFMS** specification created did fully cover all manual test cases previously created.

When creating manual test cases, priority was given to cover use cases known to be most frequently used. In A-MLC, most frequent use cases all start with a *slir/6* request and ends with a *slia/5* response with all requested presence data included, i.e., only a few test cases represented error cases. Due to the manual work involved in defining, executing and validating a manual test case only a minimal number were defined. The test suite was expected to get a reasonable coverage for a reasonable cost with the tools available. In the evaluation, these test cases were translated into a format suitable for **ERLY MARSH** so that automatic test case execution, including validation, could be applied. As with the Randomly generated test suites above, we then first generated the symbolic test suite with all symbolic test cases using the original base model (All-Paths^{org}), and then selected matching abstract test cases.

7.2.3 Test execution strategies

One of the most important factors when comparing test suites in an industrial setting is the execution time of the test suite on the SUT. This because of the often strict time constraints which limits available time for testing. We have two possible options for executing all test cases in a test suite:

1. Execute one test case at a time, and wait for the test case to finish until the next test case is executed.
2. Execute test cases concurrently, and thus not wait for a test case to finish until a new test case is executed. A consequence of such a strategy is of course that the SUT may be overloaded with traffic, that may cause it to drop requests. Any such failure was detected, so to avoid this the test client was configured with a delay between requests made. We did not try to find the optimal delay.

The A-MLC is designed to be able to handle multiple concurrent requests, thus running test cases concurrently caused no problems in general. The **ERLY MARSH** Verificator was able to include SUT logs, measurements and alarms, but not able to separate concurrent requests from each other. Thus, the degree of validation of an executed test case was dependent on the execution strategy used:

- If executing test cases concurrently, we only validated each test case against the formal model. Thus, we can only validate the order and parameters of all input and output expressions in a test case.
- If executing one test case at a time, we additionally validated resource utilization and unexpected exceptions. This required one request at a time, because of measurement tool limitations.

7.2.4 Base models

We refer to a *base model* as the model on which different test selection techniques are applied. We considered three different base models for the A-MLC ERLANG/EFSM. These were:

- The *original base model*, created from the original specification and projection properties on 4 boolean symbolic parameters.
- The *normalized base model*, created from the normalized specification (see Section 2.10), and a projection property with 4 boolean parameters.
- The *reduced base model*, was created from the original specification and a projection property with 20 (mainly) boolean parameters.

The projection properties for the original and normalized base models limits the number of parameters in the *slia/5* response and use of each MAP session to a single request/response. The additional projection properties for the reduced base model limited the number of parameters in the *slia/5* response event further and only covers test cases using the PSI presence method.

Below we sometimes use the notation $Cov^{BaseModel}$ where *Cov* is one of the coverage criteria from Section 7.2.1, and *BaseModel* is the base model from which test cases were selected. For example, $AllEdges^{org}$ refers to a test suite that cover all possible edges at least once in the original base model.

7.3 Dependent variables

The *dependent* variables are used to measure different aspects of the evaluation. The values of dependent variables are influenced by changes to the independent variables.

When executing a concrete test case we assume a black-box view of the SUT and therefore initially only observe failures. Investigating an observed failure more closely can reveal a number of faults.

There exist many ways to measure faults in software. As we assume the number and effect of the faults in the software we are testing to be unknown to us we:

1. cannot assure a single failure on a test case is only caused by a single fault, and
2. cannot assume continued execution after a fault do not cause additional failures that would not have been present without the initial fault.

One approach could be to remove faults one by one until no faults can be found for each test case. Here, we instead count *bug fixes*, each of which is a correction of a fault detected, see Section 7.3.2. Our approach was to correct all faults with bug fixes, then the number of bug fixes exposed was counted. Thereby getting a fairly good picture of the number of faults detected by the test suites. The source code coverage of the implementation was measured after all faults were corrected with bug fixes.

In this evaluation we will measure the dependent variables; (1) number of failures detected, (2) number of bug fixes required to remove all faults causing the failures, (3) source code coverage, (4) size of abstract test suites, and (5) execution time (for test suite generation and execution).

7.3.1 Failures

We here define a *failure* to be the observed inability of a SUT to perform a request for some reason. Failures were categorized by the origin of the failure as follows:

Unexpected output event types

While executing a test case we observe event types not expected according to the specification. This can be further categorized into:

1. *unexpected output* that occurs when the test execution environment does not expect to receive any more output event, but still receive an additional output event,
2. *bad output* that occurs when the test execution environment expects to receive one output event type, but receives another output event type, and
3. *missing output* which occurs when the test execution environment expects to receive an output event type, but execution stopped too early and the output event was never received.

All comparisons were made after parsing concrete output events and generating corresponding abstract events, on which a comparison was made with the expected trace generated from the formal model. In all cases the trace before the occurrence of the failure was verified to be correct.

Unexpected output event parameter values

While executing a test case we observe correct event types, but unexpected event parameter values according to the specification. We here

chose to only define two categories (based on a status code in the *slia/5* event) because they were all considered to have a low severity level, and the cluttering it would imply to include all possible sub-categories. However, the **ERLY MARSH** tool was able to find many more sub-categories by combining sets of parameters.

Resource utilization

It was assumed that a test case executed on the SUT should not cause excessive memory consumption, as that would indicate a memory leak. Thus we define one failure to be an observed bad resource usage. This was further categorized in: (1) *hanging processes* which occur when, after the test case has finished, additional processes are running compared to the number of processes running before the test case started. Processes created by SUT activities, e.g., to perform backup of data, were identified and filtered out not to influence the results. (2) *atom table leak* occurs when additional memory for the atom table was used for a test case. Note that this is a problem since the current implementation of **ERLANG** does not garbage collect the global atom table that holds all **ERLANG** atoms used. If we can measure an increased size of the atom table we are likely to have a memory leak. However, a first use of an atom always causes allocation of memory but such an allocation is considered harmless as long as it only occurs the first time a test case is executed. Thus, when detecting a suspicious memory leak the test case was executed a second time before reporting an atom table leak failure.

Exceptions

The **ERLANG** run-time system may generate an exception when a run-time error is detected. Typically, A-MLC logs data output from such exceptions. Thus, after the test case had finished these logs were filtered for known keywords associated with exceptions. Examples of such keywords generated by the **ERLANG** interpreter are 'EXIT' (the special atom used to help represent exceptions in the **ERLANG** run-time system), `function_clause` (a matching function clause could not be found), `case_clause` (a matching case clause could not be found) and `bad_arg` (an expression was given an argument it could not handle).

7.3.2 Faults

We here define a *fault* to be a cause of a failure. Each fault may cause multiple failures, and each failure may be caused by multiple faults. Thus we can say that the existence of a failure indicates a need to fix one or more faults. We introduce *bug fixes* as a way to characterize faults.

The SUT was not seeded with any faults. All faults found are “real” faults, not (yet) detected by any other kind of testing. A main pur-

pose of the testing presented here is to discover faults on a functional level, captured in the A-MLC `ERLANG/EFsm` specification of the SUT. Such faults typically originate in requirements misunderstood or not covered by the implementation. With respect to finding badly implemented requirements, it is our belief that counting real faults gives more accurate results, than mutating the implementation with, what is expected to be, common programming errors.

In our experiments we wish to study programs containing *multiple faults*. This causes additional obstacles as the faults may be dependent. Thus we cannot decide which faults a particular test case reveals as detection of a particular fault may be dependent on the existence of some other fault. We avoid this by measuring number of different bug fixes used to remove the faults.

Correction of faults (bug fixes)

Fault-detection effectiveness is the ability of a test suite to detect faults. It can be measured by studying programs containing known faults, and count the numbers of faults detected for each test suite. Since we want to count all possible faults discovered by a test suite we run test cases on corrected source code. The number of bug fixes needed to remove the failures, roughly corresponds to the number of faults detected. In order to ensure a close relationship between faults and bug fixes we need to exactly define how a detected fault can be corrected with bug fixes.

A fault must originate in some executable source code. We define an *executable statement* to be an `ERLANG` expression, such as a matching or function call, that may be executed when the program is executed. Consider a sequence of executable statements s_1, \dots, s_n where a fault can be attributed to statement s_k . We define a bug fix to be a change of s_k into a correctly behaving `ERLANG` expression. A bug fix that requires the addition of an `if` or `case` expression is considered to be as many bug fixes as there are clauses in the `if` or `case` expression with mapping functions to correct `ERLANG` expressions other than the identity function.

Example 7.2 Consider the following extract from an `ERLANG` program:

```
Var1=1, Var2=2,
```

where `Var2=2` is considered a faulty statement as the value matched with `Var2` depends on a variable `SunnyDay`. Thus, we introduce `if` clauses for the possible values of `SunnyDay`. The resulting `ERLANG` program includes an `if` expression such that we get

```
Var1=1,
Var2=if
    SunnyDay==yes ->    two;    % bug fix 1
```



```

    SunnyDay==maybe -> three; % bug fix 2
    SunnyDay==no -> 2 % NO bug fix
end,

```

As the replacing if expression has two additional possible execution paths we say we have made two bug fixes. \square

When comparing test suites in this evaluation, all test suites were executed without any faults detected, as they had been replaced with bug fixes. To identify and count bug fixes each bug fix was additionally annotated with a call to a unique bug fix function. Thus, in the evaluation the example above would become

```

Var1=1,
Var2=if
    SunnyDay==yes -> bug1(),two; % bug fix 1
    SunnyDay==maybe -> bug2(),three; % bug fix 2
    SunnyDay==no -> 2 % NO bug fix
end,

```

where the implementation of `bug1()` and `bug2()` allowed us to implement counters etc. independently from the SUT.

7.3.3 Source code coverage

We may also measure effectiveness of a test case selection by studying source code coverage. We are here limited to usage of existing, available tools and are only aware of the `COVER` [Erl 15] tool, part of the `ERLANG/OTP` distribution, and a more capable prototype tool described in [Widera 04]. Due to the beta status of the Widera tool we decided to use the `COVER` tool to count how many times each executable line of source code was executed when a program was executed. Thus, by examining output from the coverage tool `COVER`, we could e.g., directly see exactly how many times each bug fix function had been used (i.e., number of times corresponding bug fix was applied) by a test suite.

The `COVER` tool counts how many times each executable line of source code is executed when a program is run. An executable line contains an `ERLANG` expression such as a matching, guard or a function call. A blank line or a line containing a comment, function head or pattern in a `case-` or `if` expression is not executable. Thus, for example,

```

foo() ->
    Var1=2,
    Var2=two,
    if
        SunnyDay==yes,

```

```

    HappyDay==yes ->
        Var1;
    SunnyDay==maybe,
    HappyDay==yes ->
        Var2
end.

```

will be counted for as 4 possible executable lines that may be covered in a test suite.

7.3.4 Abstract test suite size

The number of possible abstract test cases that may be created from a single symbolic test case is found by instantiating the symbolic parameters in all possible ways such that the path condition evaluates to **true**. Naturally this number depends on the symbolic parameters that occur in the path condition. But it also depends on those symbolic parameters that occur in the trace but *not* in the path condition. This because the actual implementation of the SUT may very well depend on the corresponding concrete parameters and constants.

We assume that all symbolic parameters that occur in a symbolic test case must be considered - even if not used in any expression. This includes all input expression parameters in the symbolic trace and configuration parameters depending on those input expression parameters. Further dependencies between symbolic parameters were not considered. The total number of abstract test cases, represented by a single symbolic test case, can therefore be calculated by counting how many different ways all these symbolic parameters can be assigned values while satisfying the path condition. The number of possible abstract test cases that may be created from a symbolic test suite is the sum of all abstract test cases that may be created from each symbolic test cases in the symbolic test suite.

7.3.5 Execution time

We consider both the time to generate a test suite and to execute a generated test suite. The time to generate a test suite involves both the time to generate a symbolic test suite and an abstract test suite. The execution time of a concrete test suite against a SUT is measured in a test environment. We consider both sequential execution and concurrent execution of test cases against a SUT.

7.4 Threats to validity

All controlled experiments are subject to threats to validity that must be considered when evaluating the results, see e.g., [Briand 07]. Here we further distinguish between:

internal validity i.e., whether the outcome of the evaluation depends only on the independent and dependent variables discussed in Section 7.2 and Section 7.3. I.e., “Did the experimental treatments make a difference?”

external validity i.e., whether the results of the evaluation can be applied to circumstances outside the specific settings in which the study was carried out. I.e., “To which extent can the results be applied to other systems in need of testing?”

7.4.1 Internal validity

The combined test case generation tool used, **ERLY MARSH**, is a novel tool that has been developed in parallel with this evaluation. Although a lot of effort has been put into ensuring correctness of the tool, its maturity level is a risk. It should be noted that we can not use **ERLY MARSH** to validate itself.

Fault detection is limited to what can be measured. We consider all output generated by the SUT, including; generated output events, alarms, counters and logs. But automatically detecting faults is a difficult problem to solve completely. Alarms may not be raised as they should. Logs, not designed to be automatically examined, may be scanned for the wrong keywords indicating a fault. There is also no guarantee that faults, only detectable from logs actually are logged, thus making it impossible to detect the fault. However, when verifying executed test cases we take a black-box view and only mandate correctness of (abstract) output events.

We did not exactly measure faults, but bug fixes that sometimes may be done in multiple ways. Although some care was made to create a one-to-one mapping between faults and bug fixes, we may have fixed more than one fault or introduced additional new faults. Also, the existence of multiple faults makes it possible that faults are masked by other faults. Correct implementation of bug fixes would however eliminate this problem.

7.4.2 External validity

Only a single formal model and SUT was considered, consequently limiting the validity in this evaluation. Several candidates for other studies exist, the only reason these has not been included are time constraints. The bottleneck is the creation of a formal model of the SUT, which re-

quires significant time and insight into the SUT. The SUT chosen, A-MLC; (1) has a rather large set of possible use cases, (2) was considered rather mature as it had been deployed at customers. Thus, most simple and obvious bugs had been detected and removed. It is therefore our belief that A-MLC is well representative for the class of applications we are mainly interested in. In addition, it also operates in a standardized environment and therefore shares much functionality with other nodes operating in a similar environment (GSM/UMTS core network).

As the formal model and implementation of A-MLC were created separately from each other they have different structures. This has the implication that certain selection mechanisms may be favored in code coverage. We can only avoid this by running all possible abstract test cases or creating a formal model with a well defined mapping directly to the implementation. An implementation more similar to the formal model in structure may thus reveal different results.

When comparing symbolic test case selection techniques we did not consider different search strategies. We only used a depth first search strategy. As the search strategy directly influences which test cases are selected by the observer, we could have generated different test suites with the use of alternative search strategies. This could have influenced the results.

We use “real” faults to measure performance of the test selection. Although it may be seen as a strength to be able to detect faults not detected in any other testing performed, this may simply be because the SUT was poorly tested. It was e.g., known that parts of the SUT was more “well-tested” than other parts. Not having control of the failures also gives us no control in distributing faults in an even manner. This may affect the results, especially for those coverage criteria that only generated small test suites. When the test suites were applied A-MLC was a relatively mature product which may also affect the types of faults found.

Another performance measure used is implementation code coverage, by the `COVER` tool. However, this tool only measures line coverage limiting the usefulness of the results returned.

8. Results using **ERLY MARSH** on A-MLC

This chapter presents the results of our case study, whose setup was presented in the previous Chapter 7, for evaluating our framework for model-based testing on A-MLC (Advanced Mobile Location Center) and different strategies in model based testing. For specification of A-MLC we use an **ERLANG/EFSM** specification (see Section 2), and for definition of coverage criteria we use observer automata (see Section 3). The **ERLY MARSH** tool set (see Section 6) is used to generate and execute all test suites.

The chapter is organized as follows. Section 8.1 summarizes the results of the experiments described in Chapter 7. Section 8.2 further discusses the results concerning measured failures, Section 8.3 discusses the results concerning measured faults, Section 8.4 discusses the results concerning measured code coverage, and Section 8.5 discusses the results concerning measured size of test suites and execution times. Finally, in Section 8.6 we summarize and discuss our findings.

8.1 Summary of the results

Our goal was to study the effects of automatic test suite generation based on different techniques for selecting test cases. We did this by studying the effects on the dependent variables (i.e., Failures, Faults, Source code coverage, Abstract test suite size, and Execution time) while generating and executing test suites with four symbolic test case selection techniques, five abstract test case selection techniques, two test execution strategies, on three different base models. The results are summarized in Table 8.1, Table 8.2, Table 8.3 and Table 8.4. In all these tables we use a common terminology where

- **Failures** is the number of different failures identified,
- **Bug fixes** is the number of bug fixes exposed to correct detected faults,
- **Code Coverage** is the fraction of lines of code covered on selected **ERLANG** modules on the SUT, after excluding code out of scope from the A-MLC **ERLANG/EFSM**,
- **Abstract TC** is the maximum number of possible abstract test cases in the corresponding symbolic test suite. That is, number of ways the symbolic test suite can be instantiated. **Test Cases** is the number of concrete test cases *executed*,

- **Time gen symb** is the time measured to generate the symbolic test suite,
- **Time gen abs** is the time measured to generate the abstract test suite, given a symbolic test suite,
- **Time exe seq** is the time measured to execute the test suite sequentially, and
- **Time exe con** is the time measured to execute the test suite concurrently.

Further terminology used in the tables, explanations and motivations of the test suites used in the case study are given as follows. Section 8.1.1 further explain Table 8.1 on the coverage based test suites, Section 8.1.2 further explain Table 8.2 on the random test suites, Section 8.1.3 further explain Table 8.3 on the manual test suites, and Section 8.1.4 further explain and motivates the test suites based on the reduced base model in Table 8.4.

8.1.1 Coverage based test case selection

Using the original and normalized base models, see Section 7.1.2, we first generated symbolic test suites considering the coverage criteria All-Locs (*All target locations* Section 4.1.2), All-Edges (*All edges* Section 4.1.1), Def-Use (*Definition-Use pair* Section 4.1.4), and All-Paths (*All paths* Section 4.1.3). We then created abstract test suites from the generated symbolic test suites using **Biased** and **Rndsym** abstract test case selection, see Section 7.2.2.

A summary of the results, using test suites generated using observer automata, is shown in Table 8.1 where **Symbolic TC** is the number of symbolic test cases in the test suite. Furthermore, here we only select a single abstract test case, and create a single concrete test cases from each symbolic test case. Thus, the number of *executed* concrete test cases is identical to the number of symbolic test cases in the test suite. Note that results for **Biased** and **Rndsym** are shown in different rows (e.g., **Bug fixes**/**Biased** and **Bug fixes**/**Rndsym**) whenever a significant difference was measured. For **Failures**, **Time gen abs**, **Time exe seq** and **Time exe con** we did not measure any significant difference between **Biased** and **Rndsym**. Thus, only the results for **Rndsym** abstract test case selection is shown for these measures.

8.1.2 Random test case selection

To better understand the power of applying the coverage based test suite generation techniques we also created a number of Random test suites, see Section 7.2.2. The results of running these test suites are shown in

Original base model:

Test suite name:	All-Locs ^{org}	All-Edges ^{org}	Def-Use ^{org}	All-Paths ^{org}
Symbolic TC	5	422	6,292	82,423
Abstract TC	31104	9 343 336	710 897 147	400 781 346 067
Failures/Rndsym	2	12	12	15
Bug fixes/Biased	13	78	80	93
Bug fixes/Rndsym	13	78	83	96
Code cov/Biased	58.18%	94.36%	92.67%	95.68%
Code cov/Rndsym	59.21%	95.49%	94.27%	97.18%
Time gen symb	00:00:17	00:00:19	01:18:23	00:10:57
Time gen abs/Rndsym	00:00:01	00:00:01	00:00:10	00:13:26
Time exe seq/Rndsym	00:00:04	00:06:50	03:54:23	62:00:40
Time exe con/Rndsym	00:00:01	00:00:12	00:03:18	00:55:33

Normalized base model:

Test suite name:	All-Locs ^{nor}	All-Edges ^{nor}	Def-Use ^{nor}	All-Paths ^{nor}
Symbolic TC	5	310	4 275	59 209
Abstract TC	641 520	14 369 363	770 390 937	400 781 346 067
Failures/Rndsym	2	12	12	15
Bug fixes/Biased	12	77	78	91
Bug fixes/Rndsym	13	73	81	95
Code cov/Biased	57.33%	91.73%	91.73%	94.55%
Code cov/Rndsym	58.08%	93.52%	94.36%	97.37%
Time gen symb	00:00:13	00:00:13	00:28:00	00:04:36
Time gen/Rndsym	00:00:01	00:00:01	00:00:07	00:07:16
Time exe seq/Rndsym	00:00:05	00:06:00	02:49:54	52:50:04
Time exe con/Rndsym	00:00:02	00:00:08	00:02:18	00:35:16

Table 8.1. Summary of test results for coverage based test case selection, using the original and normalized base models for A-MLC.

Test case weight:

Test suite name:	Rnd_{1k}^{tcw}	Rnd_{12k}^{tcw}	Rnd_{48k}^{tcw}	Rnd_{100k}^{tcw}
Test cases	1000	12 000	48 000	100 000
Symbolic TC	968	8 464	20 276	28 583
Failures	12	13	15	15
Bug fixes	48	53	57	60
Code Coverage	83.08%	86.84%	89.13%	90.76%
Time exe con	00:00:46	00:09:02	00:31:59	01:02:00

Event weight:

Test suite name:	Rnd_{1k}^{ew}	Rnd_{12k}^{ew}	Rnd_{48k}^{ew}	Rnd_{100k}^{ew}
Test cases	1000	12 000	48 000	100 000
Symbolic TC	313	1195	2361	3283
Failures	13	15	15	15
Bug fixes	83	98	98	98
Code Coverage	96.20%	97.58%	97.55%	97.79%
Time exe con	00:00:14	00:02:56	00:11:01	00:23:55

Table 8.2. Summary of test results for creating and executing a selection of the random test suites on the A-MLC.

Table 8.2, where Rnd_i^w is a random test suite utilizing weight w (i.e., test case weight ew or event weight tcw) with i test cases, **Symbolic TC** is the number of different symbolic test cases covered by the random test suite.

All reported results with the random test suites are the mean values of three independently created test suites. Random test suites were created in several different sizes, for brevity only a subset are shown in Table 8.2.

8.1.3 Manual test case selection

At the time the evaluated snapshot of A-MLC was made, not all of the traffic scenarios found in the functional description had been deployed at a customer. Therefore, the test suite used at Mobile Arts for testing A-MLC was limited to 20 test cases, and covered 6 out of 11 traffic scenarios defined in the functional specification, representing the most frequently occurring use cases. All test suites were manually created, see Section 7.2.2, and we will refer to this limited test suite as Man_{lim} . Manual test cases for the additional traffic scenarios, were constructed by the first author following the same patterns as in the Man_{lim} test suite. The full manual test suite is referred to as Man . Only a few of these test cases represented error cases.

Test suite name:	Man	Man _{lim}
Test cases	50	20
Failures	2	2
Bug fixes	31	15
Code Coverage	80.98%	66.70%
Time exe con	00:00:02	00:00:01

Table 8.3. *Summary of test results on the A-MLC for manual test suite.*

The results of running these tests suite are shown in Table 8.3. Note that both manual test suites found the same two failures (fail₁₀ and fail₁₃), both estimated to be of low severity. Failures and severity levels are further discussed in Section 8.2.

8.1.4 Testing with projected specification and reduced validation

While executing the test suites, we were initially surprised by the large number of bug fixes exposed. After investigations, we concluded that the main reason was that the SUT was insufficiently tested since only a subset of the implemented features were deployed and used by customers.

Previous testing of the SUT at Mobile Arts had been driven by Acceptance testing (see Section 1.1) and had been concentrated to limited parts of the SUT. Other parts of the SUT had been implemented, but incompletely tested. Not only was the number of test cases in the manually selected test suite limited, but the SUT had been tested only by a developer and the values of several parameters in output events had not been validated during testing.

In order to search for failures in the more “well-tested” part of the SUT, a *reduced base model* was created from the original specification, applying a projection property with 20 symbolic parameters (mainly booleans). All set to fixed values such that the resulting model excluded features incompletely tested, such as an additional position method. Additionally, validation of parameters in output events was relaxed to validate only a subset of the parameters. The manually selected test suite, Man^{red}, contains the subset of Man which could be generated from the reduced base model. The All-Paths^{red} test suite has full coverage of all symbolic test cases in the reduced base model where a single abstract test case was selected using **Rndsym** abstract test case selection.

A summary of the results running these test suites is shown in Table 8.4 where **OK Test Cases** is the number of concrete test cases executed and validated successfully without exposing any bug fixes.

Test suite name:	Man ^{red}	All-Paths ^{red}
Test Cases	20	5817
Failures	1	7
Bug fixes	16	41
OK Test Cases	19	196
Code Coverage	66.70%	66.33%

Table 8.4. *Summary of test results found with test suites based on a reduced base model, and using limited validation.*

The manually selected test suite Man^{red} revealed only a single failing test case. As the SUT was a snapshot of a previously deployed system, with some additional development, it was expected to only reveal a few (or none) failing test cases when executing a similar test suite as had been executed during previous acceptance testing. More surprising was the large number of failing test cases when running test cases from all symbolic traces in All-Paths^{red}. To some extent these results can be explained by the fact that an inconsistent behavior for similar test cases earlier had not been tested. In the specification some effort was made to have a consistent behavior between input expression parameters and output events which caused a number of additional observed failures.

8.2 Failures found while testing

As explained in Section 7.3.1, a failure is defined to be the observed inability of a SUT to perform a request for some reason. During test suite execution a number of failures were revealed, assisted by the ERLY MARSH tool, the results were examined more closely after executing the test suites. To better understand the identified failures, they were further classified into three different severity levels:

- *low*, indicating the failure was expected to have no, or only limited, impact on a SUT deployed in a live network, without demanding applications requiring correct quality of position etc.,
- *middle*, indicating the failure could have a severe impact on the SUT during operation in a highly utilized live network, and
- *high*, indicating the failure would have a severe impact on the SUT during normal operation in a live network.

Table 8.5 summarizes the results. Examining the identified failures more closely they could be further categorized as:

- *Unexpected output*: $fail_1, \dots, fail_5$. Test cases where the test execution environment does not expect to receive any more output event, but still receive an additional output event. All output event

Name	Severity	All-Locs ^{org}	All-Edges ^{org}	Def-Use ^{org}	All-Paths ^{org}
fail ₁	low		X	X	X
fail ₂	middle		X		X
fail ₃	middle				X
fail ₄	middle		X	X	X
fail ₅	low		X	X	X
fail ₆	low		X	X	X
fail ₇	middle				X
fail ₈	low		X	X	X
fail ₉	low		X	X	X
fail ₁₀	low	X	X	X	X
fail ₁₁	low		X	X	X
fail ₁₂	middle			X	X
fail ₁₃	low	X	X	X	X
fail ₁₄	high		X	X	X
fail ₁₅	high			X	X

Table 8.5. Failures found when running test suites generated from the original base model.

types towards the GSM/UMTS core network occurred as unexpected events in some test case. As over utilization of the HLR should be avoided (most of the core network depends on the HLR) failures detected with an additional output event towards the HLR was classified as a middle severity failure. These were fail₂ (unexpected first *sri/1*), fail₃ (unexpected second *sri/1*), and fail₄ (unexpected *ati/1*). All other failures (fail₁ with unexpected *fsm/3* and fail₅ with unexpected *psi/3*) were classified as low severity failures.

- *Bad output:* fail₆, ..., fail₇. Test cases where the test execution environment expects to receive one output event type, but receives another output event type. These failures occurred when an immediate position request was expected, but the A-MLC instead chose to first force an update of the position (fail₆ low severity, expected *psi/3* but got *fsm/3*) or request additional information from the HLR (fail₇ middle severity, expected *psi/3* but got *sri/1*).
- *Missing output:* fail₈, ..., fail₁₁. Test cases where the test execution environment expects to receive an output event type, but execution stopped too early and the output event was never received. This occurred for all event types towards the GSM/UMTS core network in some test case. As a result of these, the requested quality of position was not performed, although it *may* have been fulfilled if

the expected events had occurred. In more detail; in $fail_8$ a second $ati/1$ was expected after a forced update, in $fail_9$ a second $psi/3$ was expected after a forced update, in $fail_{10}$ a forced update was expected after an $ati/1$, and in $fail_{11}$ a forced update was expected after an $sri/1$. All these failures were classified with low severity level.

- *Unexpected output event parameter values: $fail_{12}, fail_{13}$.* The $fail_{12}$ failure were test cases failing by a position response code indicating that a successful $slia/5$ response was received, despite it should return an error according to the specification. This failure was classified with middle severity level. The $fail_{13}$ failure were test cases failing by other unexpected or bad parameter values in the $slia/5$ response to the user. This failure was classified with low severity level.
- *Resource utilization: $fail_{14}$.* A highly severe failure was found when executing over 1000 requests. The reason was that the XML parser did not free an allocated hash table (ERLANG ets table), causing a complete system crash after a system limit was reached. This failure had to be fixed before further testing could be made.
- *Exceptions: $fail_{15}$.* Missing clause in state machine caused the handler process to exit, and a default error response was returned.

In summary, the different test suites found different subsets of the failures and only the largest, most complete test suite (All-Paths^{org} and the larger random test suites e.g., Rnd_{48k}^{tcw} and Rnd_{100k}^{tcw}), found all. It can be noted that the (minimal) All-Locs^{org} test suite did not find any of the highly critical failures. The All-Edges^{org} test suite found almost as many failures as the Def-Use^{org} test suite, despite that test suite included 93% less executed test cases. However, it did not find the severe memory leak, $fail_{14}$, since this required the size of the test suite to be at least 1000 test cases.

8.3 Faults found while testing

The failures described in Section 8.2 were all caused by detected faults. All these faults are either found by comparing the expected behavior in the formal model with the actual behavior of the SUT, or faults that were found based on general assumptions regarding resource utilization. The formal model does not give any information regarding resource utilization. Nevertheless it was assumed that running a test suite should only affect resource utilization temporarily such that allocated resources, that could harm long-term usage, were always returned. After correction of the faults with bug fixes, the number of different bug fixes exposed when running the test suites was noted. In total we found it necessary to patch the SUT

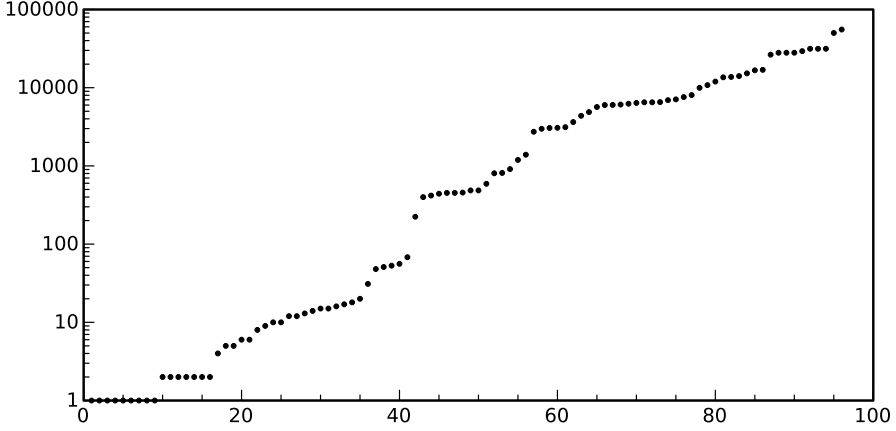


Figure 8.1. Number of test cases exposing each bug fix when running the All-Paths^{org} test suite with random abstract test case selection. The y-axis shows the number of different bug fixes exposed by the same number of test cases, and the x-axis shows the number of test cases exposing a bug fix.

with 102 bug fixes to make it compliant with the A-MLC `ERLANG/EFSM` specification. None of the test suites found all of these bug fixes. The faults in the implementation varied in how hard they were to detect.

Figure 8.1 shows how many times each bug fix was exposed when executing the All-Paths^{org} test suite. It can be noted that 9 bug fixes are only exposed by a single test case. These represent faults that can be assumed to be hard to detect and are therefore only detected by a few of the test suites in the evaluation. On the other hand, a majority of the bug fixes are exposed by more than 1000 test cases. The most common bug fix was exposed by 55446 test cases.

We then compared test suites by relating the fraction of the exposed bug fixes found with the size of the test suite measured in number of test cases. Figure 8.2 summarizes the results for the coverage based test suites. With respect to the rather few coverage based test suites and the difference among them it can be noted that choosing the original or normalized base model had an effect on the outcome, although limited. Test suites selected from the original base model in general perform better than the normalized base model. This, despite both representing the same state space. This can be understood by considering how the chosen coverage criteria interact with the “coding style” used. In the normalized base model the number of edges has been reduced which causes the generated symbolic test suites to become smaller. Our results indicate that this also reduces the quality of the generated test suites. An explanation can be that with smaller size of the symbolic test suite there are more possible ways to instantiate each symbolic test case. Another possible explanation

can be that the logical structure of the specification is of importance for which test cases are selected, but the normalized base model blurs this structure.

Random abstract test case selection (**Rndsym**) perform better than biased (**Biased**) regardless of base model. This was as expected, but the differences were quite minimal though. Therefore, we did not investigate further possible improvements, see Section 11.9. It can be noted that the influence the abstract test case selection technique has, also depends on the amount of details included in the model. A more detailed base model implies a closer correspondence with generated test suites and implementation, but also a larger state space. On the other hand, a more abstract base model increase the importance of the abstract test case selection technique used. The specification used in the evaluation can be considered to be rather detailed and close to the implementation. Thus, the test suite generation on the base models was expected to be more important than the abstract test case selection techniques.

The Man test suite did expose relatively few bug fixes. Most test cases included in the manual test suites were test cases with presence data successfully requested and responded back to the user. Thus, it can be concluded that test cases with failing requests (because of bad configuration data) and responses with bad or missing presence data only exercise a relatively small part of the source code. The manual test suite Man also exposed rather few bug fixes (30%) which can be explained by the fact that these test cases had been previously executed.

Next we ran random test suites using test case weights (Rnd^{tcw}) and event weights (Rnd^{ew} and Rnd^{sew}) where Rnd^{sew} are test suites based on the simpler event weights requiring no knowledge on which computation steps the current state in the execution enable, see Section 7.2.2.

As can be seen in Figure 8.3 the coverage-based test suites, with the exception of All-Locs, performed rather well. Of the coverage-based test suites, the All-Edges test suite stands out by a comparatively good relationship between exposed faults and test suite size. We note that the random test suite using a uniform distribution over test cases (using test case weights) performs clearly worse than all other test suites for test suites larger than a few hundred test cases; this is to be expected since its bias will leave some parts of the model unexplored. There was no clear difference in fault-detection capability between coverage-based and random test suites of similar size. It should then be remembered that the random test suites base the selection of next input on information provided by the A-MLC specification. As the size of test suites increase, the law of decreasing marginal utility sets in; larger test suites only give a small improvement in the number of exposed bug fixes. It should also be noted that both types of random test suites detected faults not detected by the manual or any of the coverage based test suites.

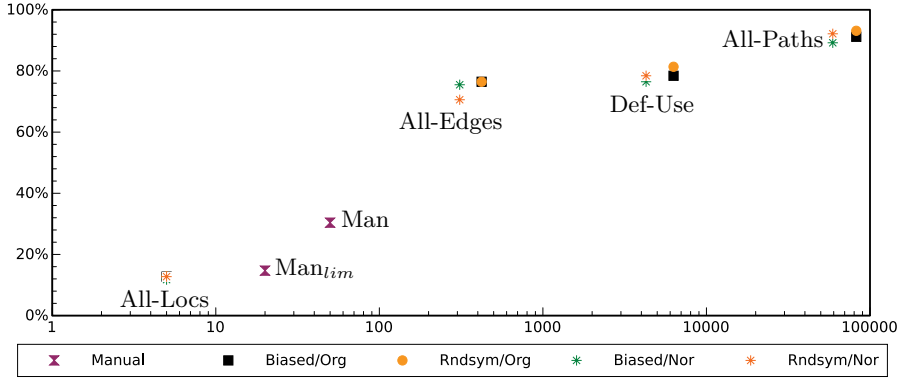


Figure 8.2. Relative number of different bug fixes exposed when running the coverage based test suites for both the original and normalized base model. For reference, the manual test suites are additionally included. The x-axis shows the number of test cases executed, and the y-axis shows the percentage of bug fixes (of all 102) that were exposed.

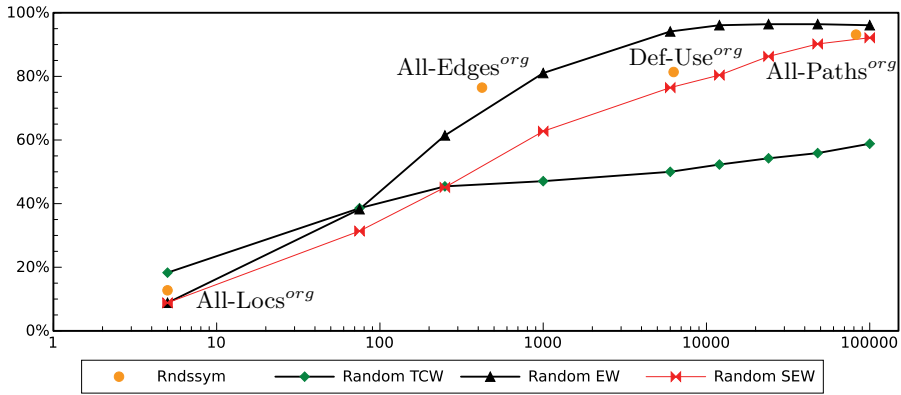


Figure 8.3. Relative number of different bug fixes exposed when running the coverage based test suites with the original base model and random abstract test case selection, random test suites generated using test case weights, and two different random test suites generated using event weights. The x-axis shows the number of test cases executed, and the y-axis shows the percentage of bug fixes (of all 102) that were exposed.

8.3.1 Characteristics of a selection of test suites

The tested version of A-MLC included some functionality only tested to a limited extent. A hypothesis was therefore that some parts of the SUT contained more faults than other parts, and because of this, certain test cases would expose significantly more bug fixes than others. Thus, a test suite with many test cases exposing many bug fixes could be suspected to better cover the less tested part of the SUT. Further, a test suite with better coverage of the less tested part could be suspected to expose more different bug fixes in total. Figure 8.4 shows the result of measuring

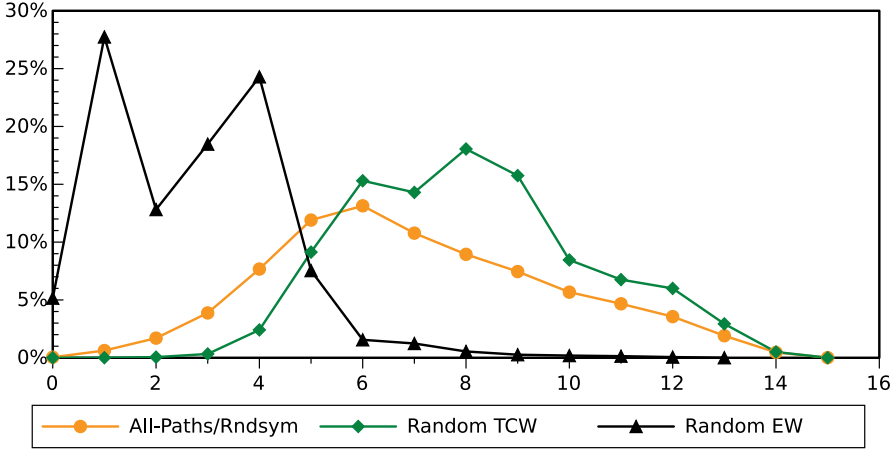


Figure 8.4. Number of different bug fixes exposed by test cases when running the All-Paths^{org} test suite with random abstract test case selection, the random Rnd_{100k}^{tcw} test suite, and the random Rnd_{100k}^{ew} test suite. The x-axis shows the number of bug fixes exposed by a single test case, and the y-axis shows the percentage of test cases in the test suite that expose the same number of different bug fixes.

the number of different bug fixes exposed by each test case part of the All-Paths^{org}, Rnd_{100k}^{tcw}, and Rnd_{100k}^{ew} test suites.

It can be noted that the Rnd_{100k}^{tcw} test suite has the highest concentration of test cases exposing many bug fixes. Still, in Figure 8.3 it does not perform very well with respect to exposed bug fixes. The reason is most likely that those faults detected on test cases with a high concentration of bug fixes are easy to detect, and found by many different test cases.

Investigating further, as can be seen in Figure 8.5, the length of test cases in the two random suites and All-Paths^{org} test suite show very different characteristics. While the random test suites using event weights have a large number of short test cases, the random test suite using test case weights have a large number of long test cases, The All-Paths^{org} test suite is more balanced although longer test cases are more frequent than

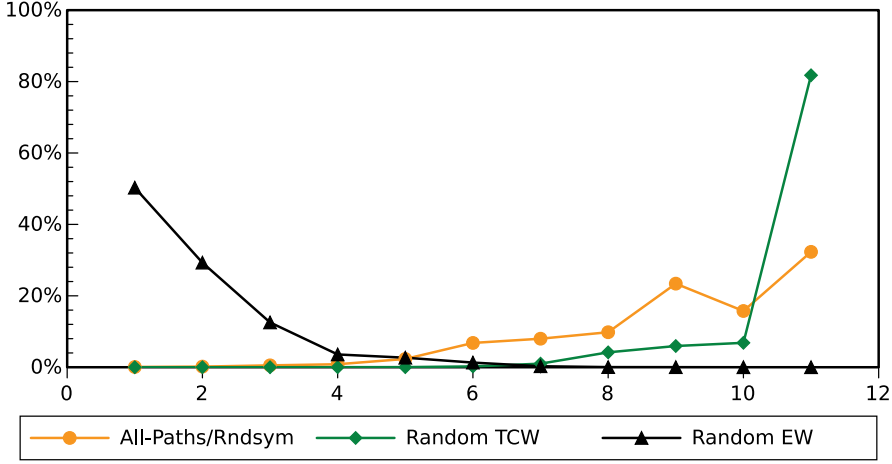


Figure 8.5. Length of test cases (number of edges) in the All-Paths^{org} test suite with random abstract test case selection, the random Rnd^{tcw}_{100k} test suite, and the random Rnd^{ew}_{100k} test suite. The x-axis shows the length of the test case, and the y-axis shows the fraction of test cases in the test suite with that length in the test suite.

shorter. The Rnd^{ew}_{100k} test suite, with many shorter test cases, has a high concentration of test cases requiring few bug fixes.

Examining the source code of the SUT it also seems a large portion of the source code can be covered by only using short test cases, as longer test cases use the same function over and over again. Thus, for this particular SUT it seems more efficient to run many different shorter test cases.

Continuing our investigation, we measured *when*, during the execution of a test suite, new bug fixes were exposed. With the characteristics of the SUT that most bug fixes are exposed by short test cases it seems natural that the execution order is of importance. In the upper Figure 8.6 all test suites were executed in the same order as they were created. In the lower Figure 8.6 the test suites were sorted before execution such that the shortest test cases were executed first. As can be seen, the results were quite different. It is particularly striking that the compared test suites behaves so differently. The All-Paths^{org} test suites find the faults much faster compared to the event weighted randomized test suite (Rnd^{ew}_{100k}). Further, both the All-Paths^{org} and Random Rnd^{tcw}_{100k} test suite expose no more new bug fix after less than 7000 test cases. On the other hand, the Random Rnd^{ew}_{100k} test suite only expose very few bug fixes before the first 53000 test cases. Reason for this is likely that there exists faults in many different parts of the SUT. The All-Paths^{org} ensure we visit all these parts, and by ordering test cases with shortest first we also visit

these parts more randomly than when not sorted. The fact that Rnd_{100k}^{ew} in total still exposes many different bug fixes can probably be explained by that there exists some “hard to find” short test cases.

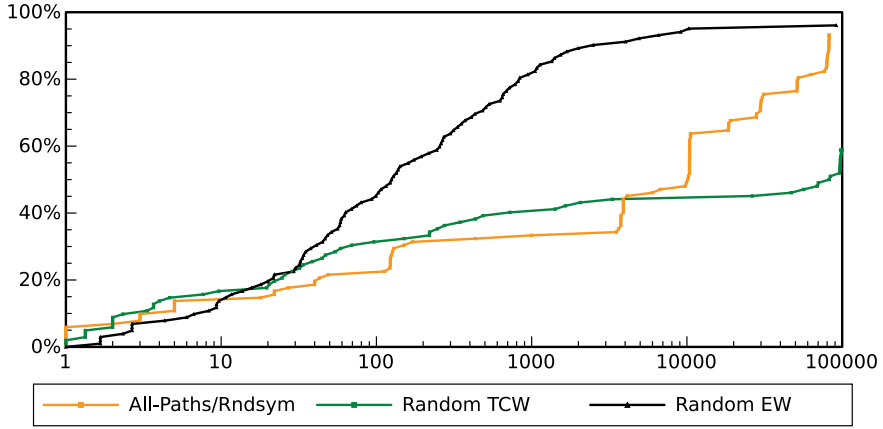
8.4 Code coverage

Code coverage of the SUT was made using the **COVER** tool on a few **ERLANG** modules. These were selected because the functionality described in the A-MLC **ERLANG**/EFSM was mainly implemented by these modules. The code coverage tool measured covered lines in the complete module, including e.g., test functions and debugging utilities.

The reported coverage was rather low at the first measurement, despite the large number of test cases executed. To get a better understanding of the effectiveness of the test suites we then examined the relevant part of the source code “by hand”. We distinguished the following 7 reasons for why source code may not be covered:

1. *Unreachable source code*: Source code that cannot be executed, regardless of input. Some of the unreachable source code were also confirmed by the **DIALYZER** tool.
2. *Unexpected exception handling*: Source code to handle exceptions originating from some unexpected fault. Handling of such potential errors (aka “defensive programming”) cannot be executed unless there is some unexpected fault in the source code. Thus, it is outside the scope of **ERLANG**/EFSM specification. As we assume the SUT to operate according to the specification (or all faults to be removed by bug fixes) such source code cannot be executed by our test suites.
3. *Diagnostic functions*: Functions not used in normal traffic cases. Typically these are functions used for debugging, test, and operation and maintenance.
4. *Implemented behavior not modeled*: These are implemented features present in the SUT that has been excluded from the formal model, but still may be used during normal operation. For example, a problem with some configuration data missing for a subscriber is always treated in the same way. It is therefore considered an isolated error case on which testing can be made by some other means. Thus, it can be noted that by extending the base model even further it would be possible to generate test suites with improved coverage.
5. *Projection property*: Source code that cannot be executed because of a projection property exclude functionality in the generated test suite.
6. *Unused abstract mapping*: Source code included by symbolic test cases part of test suite. But, we do not instantiate the symbolic test

As created:



Shortest first:

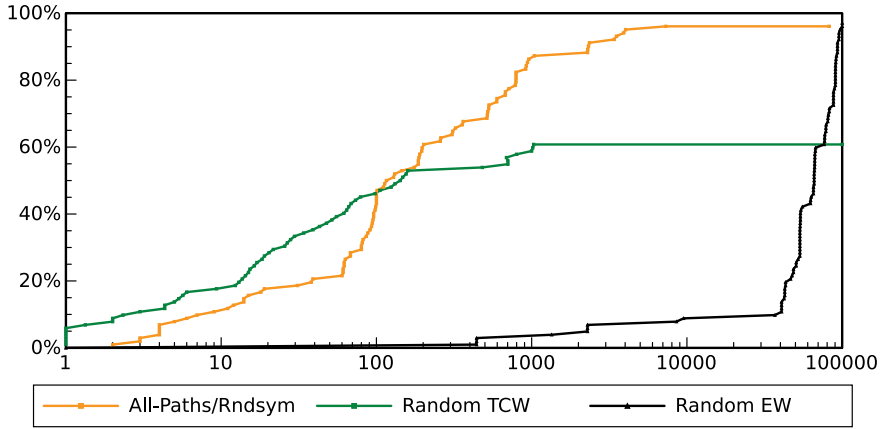


Figure 8.6. Accumulated bug fix usage when executing the test suites as created and sorted such that the shortest test cases are executed first. Results shown for the All-Paths^{org} test suite with random abstract test case selection, and two random test suites with 100,000 test cases using event weights (Rnd^{ew}_{100k}) and test case weights (Rnd^{tcw}_{100k}). The x-axis shows the number of test cases executed, and the y-axis shows the accumulated percentage of bug fixes that were exposed.

cases with all possible abstract values. Thus there is a possibility we pick abstract values in such a way source code is not executed.

7. *Unused concrete mapping*: Source code not excluded by abstract test cases part of test suite. But, each abstract parameter is mapped to a concrete value. Thus there is a possibility we pick concrete values in such a way source code is not executed.

We concluded that reasons 1 to 5 are reasons outside the scope of the test suites generated and excluded all those lines in the results reported on source code coverage. Examining the output of **COVER** after executing the test suite with the reported highest coverage (original base model, All-Paths^{org} and **Rndsymb**) we got the results in Table 8.6 for three main **ERLANG** modules in the SUT.

Coverage	Mod ₁	Mod ₂	Mod ₃
Unreachable source code	13.19%	-	-
Unexpected exception handling	2.63%	-	-
Diagnostic functions	4.40%	-	-
Implemented behavior not modeled	20.22%	42.86%	33.98%
Projection property	5.38%	5.53%	11.13%
Not part of A-MLC ERLANG /EFSM model	45.71%	48.39%	45.12%
Unused abstract mapping	1.76%	-	4.30%
Unused concrete mapping	3.30%	-	0.39%
Part of A-MLC ERLANG /EFSM model, but not covered	5.06%	-	4.69%

Table 8.6. *Classification of executable statements reported not covered by the tool COVER. The percentages shown are the part of the complete module not covered because of the given reason.*

In Figure 8.7 and Figure 8.8, the results on source code coverage are summarized. These results only include source code not covered because of Unused abstract mapping (reason 6) and Unused concrete mapping (reason 7).

Comparing the results for the different test suites, the manually created test suites have a relatively good coverage of the source code. This is as expected as the test cases part of this test suite has been selected with coverage of the main features in mind.

For comparison, in Figure 8.8, we also show the results running test suites based on the simpler event weights requiring no knowledge on which computation steps the current state in the execution enable (Random SEW), see Section 7.2.2. Figure 8.9 shows the relation between exposed

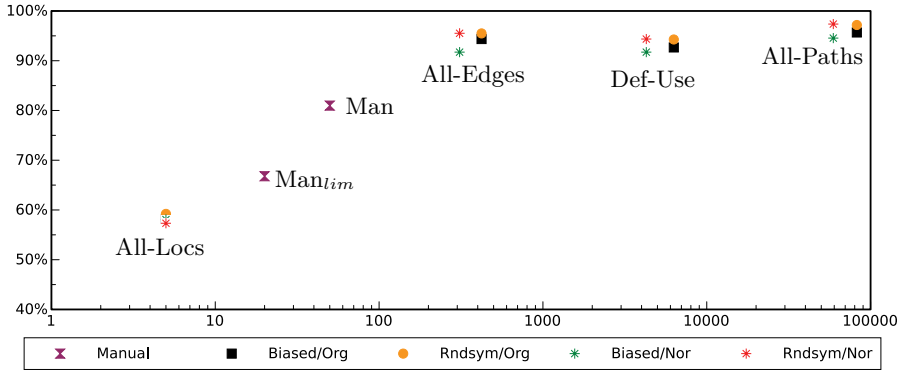


Figure 8.7. Relative source code coverage when running coverage based test suites for both the original and normalized base model. For reference, the manual test suites are additionally included. The x-axis shows the number of test cases executed and the y-axis shows the measured relative coverage.

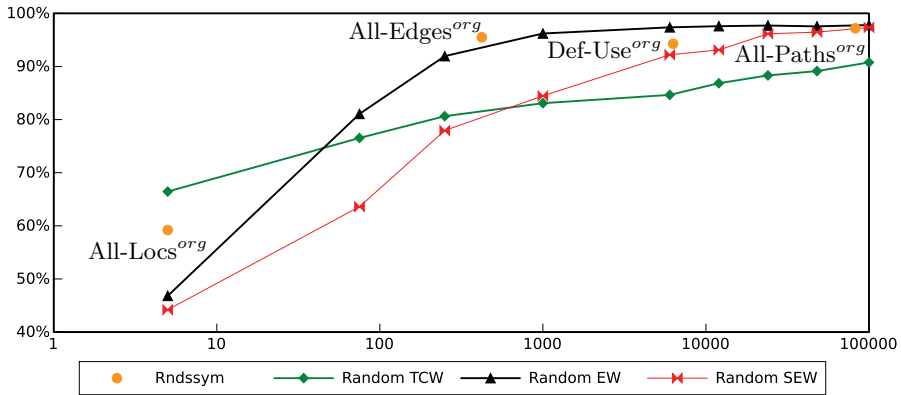


Figure 8.8. Relative source code coverage when running the coverage based test suites with the original base model and random abstract test case selection, random test suites generated using test case weights, and two different random test suites generated using event weights. The x-axis shows the measured relative coverage and the y-axis shows the number of test cases executed.

bug fixes and source code coverage. As expected, there exists a correlation such that test suites with large source code coverage also expose more bug fixes. In general it is also the larger test suites that perform best, although both Def-Use^{org} and All-Edges^{org} perform comparatively well with rather limited sizes of the test suites.

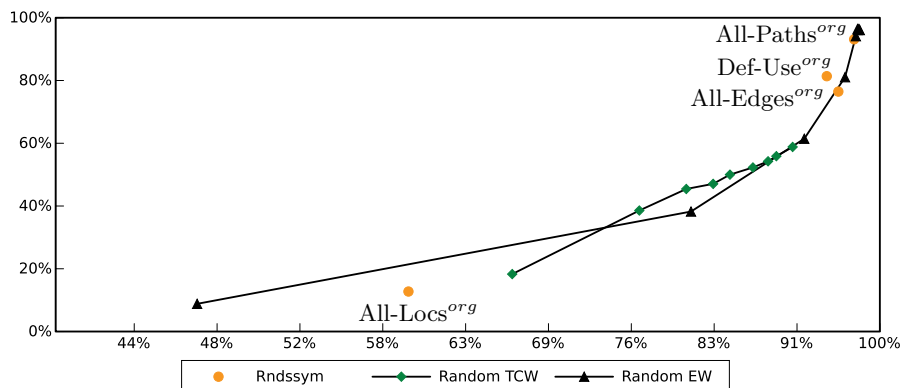


Figure 8.9. Relation between source code coverage and number of exposed bug fixes. with the original base model. The x-axis shows the measured relative coverage and the y-axis shows the percentage of exposed bug fixes.

It can be noted that the chosen base model did not affect the outcome in a significant matter. Nor did the coverage based abstract test case selection technique have any significant importance. Test suites created completely randomly in general performed worse than those selected by some criterion. As could be expected the Man test suite had a relatively high source code coverage in relation to the limited number of test cases.

8.5 Test suite size and execution times

Normally, in an industrial environment only a strictly limited amount of time can be spent on testing. The amount of time spent generating and concurrently executing the coverage based test suites once is shown in Figure 8.10. Generating All-Paths^{org} is rather efficient, but then we are not using an observer. However, in a test suite such as Def-Use^{org} , requiring a more complicated observer, the overhead in the current implementation of **ERLY MARSH** is rather significant.

The two random test suites had the same number of test cases, but the execution times differ quite significantly because of different lengths of the involved test cases.

The main reason for using selection techniques to generate (smaller) test suites with decent performance is to save test suite execution time.

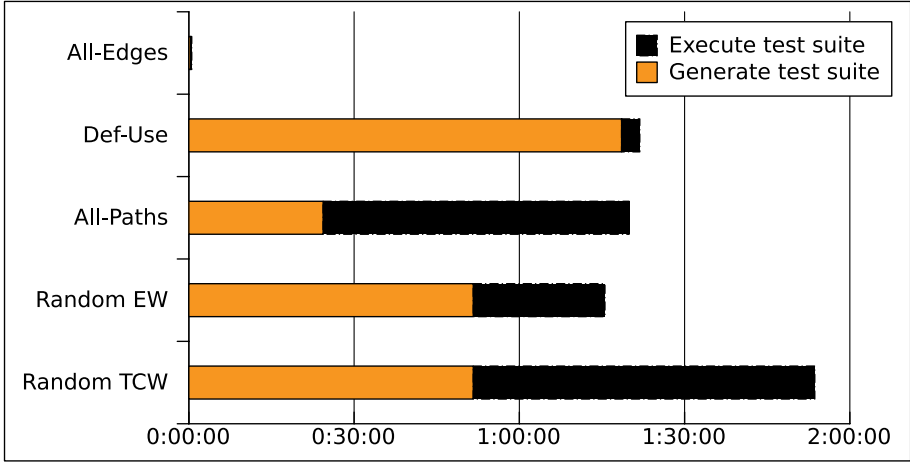


Figure 8.10. Comparison in time generating and concurrently executing the $All-Edges^{org}$, $Def-Use^{org}$, $All-Paths^{org}$, Rnd_{100k}^{tcw} , and Rnd_{100k}^{ew} test suites.

Additionally, if it is possible to run test cases concurrently this would normally be faster than running them sequentially. This is especially true for test cases including timers that need to timeout before the test case can continue. In test suites for A-MLC this is quite common. Figure 8.11 shows the difference in execution time between executing test suites concurrently and sequentially on A-MLC.

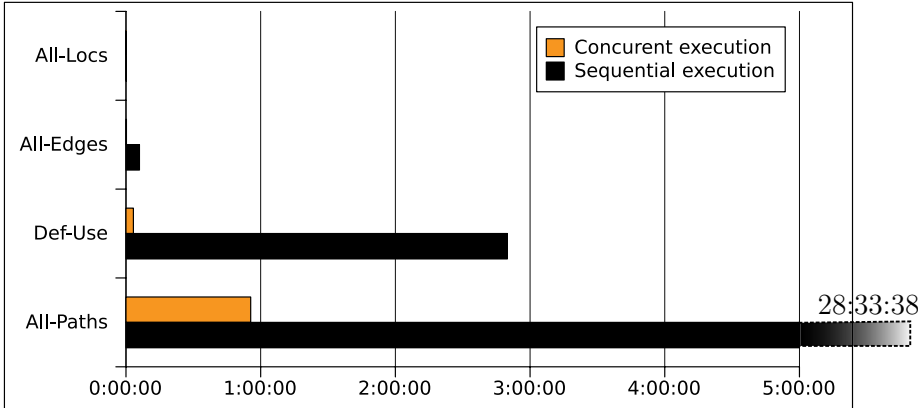


Figure 8.11. Comparing executing test suites concurrently and sequentially.

A deployed A-MLC will have many concurrent requests running. Thus, in this sense, by running test cases concurrently the SUT operates in a more realistic environment. It also makes it possible to generate higher loads on the SUT, with the additional benefit that potential performance bottle-necks can be detected. The higher load may also cause the SUT to

handle test cases incorrectly, that would not be present with some lower load.

8.6 Summary of main findings

Let us here summarize main observations and findings from the results of the case study.

- The test suite All-Edges^{org} had a comparatively good relationship between number of exposed faults and test suite size; a stronger coverage criterion increases the fault-detection capability marginally.
- The choice of base model, i.e., the structure of the specification of the SUT, had an effect of the outcome, although limited.
- We found no clear difference in power between coverage-based test suites and the randomly generated test suites with event weights of the same size, as measured by the number of detected faults and source code coverage.
- Some of the bug fixes hardest to find were only be exposed by a single symbolic test case with certain instantiations of the symbolic parameters. Thus, as it turned out, the randomly generated test suites had an advantage in that multiple test cases could be selected from a single symbolic test case. It may therefore be possible to improve the results for the coverage-based test suites, by an improved abstract test case selection, or by randomly selecting several test cases from each symbolic test case.
- As the size of test suites increase, the differences between different test suites decreases.
- Traditional manual testing often does not check that test outputs correspond completely to requirements. Formalizing requirements by a model, and using it for automated testing, can reveal deficiencies of earlier, less thorough, testing. We note that this observation need not generalize to other contexts.
- Generating a random test suite can be expensive if there exist many complicated constraints on parameters that decide whether a parameter combination is included in the model or not. In such cases many random selections of parameter values will be unfeasible test cases, not covered by the model. In our case, first randomly assigning values to all parameters, and then validating that the selected parameter value combination is feasible, turned out to be very expensive. Reason was that only 1 of approximately randomly chosen 8000 combinations of the symbolic parameter values corresponded to a valid test case. To solve this, we based all our random test suites on an existing coverage based test suite All-Paths^{org}, and used weights to generate the randomization. It can further be noted

that also combinatorial testing techniques (see Section 1.5.3) suffer from similar problems.

- During the evaluation, we found that tool support is essential for effective test suite generation, test execution, and model validation. The observation that at least one severe failure was dependent on the size of the test suite indicates that testing functional requirements by model-based testing should be complemented with other types of testing for finding resource and other non-functional errors, such as memory leaks.

8.6.1 Experiences from the Case Study

In general, we found the new specification language well suited for specification of the expected behavior of the SUT. As the formal model grew larger, we soon had problems with state space explosion. This called for various solutions on how to reduce the state space for this particular case study (by the use of abstractions, projections, symbolic execution, efficient representations of conditions etc.). Still, even with a limited state space, the number of possible test cases was found to be too large. Reducing the number of test cases to be part of such a limited test suite, called for a need to generate several different smaller test suites that could be compared. Further, characteristics was unknown before generating and executing these test suites, but was expected to depend on both formal specification and SUT. This also motivates the need for flexible test case selection.

As with any model-based testing approach, creating a high quality formal specification is a major burden. Our approach, creating the A-MLC `ERLANG/EFM` formal specification by hand, was no exception and was by far the most time consuming part of the test suite generation. Still, this (high) cost needs to be compared with the cost of writing large test suites by hand that are difficult to maintain with changing requirements. Possibly, a good approach is to start with a small specification and let it incrementally grow. Whenever it grows large enough so that all test cases can not be executed - use some selection technique.

An initial problem implementing the model in `ERLY MARSH` was how to represent the path condition and symbolic trace for each test case efficiently. NDD:s were chosen, after a first (naive) approach to represent them as symbolic `ERLANG` terms had failed because of the size of the A-MLC `ERLANG/EFM`. While performing the case study we found faults in the implementation, specification and test tool. For each fault found, test suites were required to be executed again, after a bug fix had been implemented. For faults found in the specification and `ERLY MARSH`, we additionally had to regenerate test suites again. Thus it was essential to

be able to have tool support for automatically generating, executing and validating test suites. We noted that execution times of larger test suites may become a problem. A possibility to execute tests cases concurrently was therefore essential for the test suite to hold time constraints, even if this meant validation had to be relaxed. Concurrent execution also put an additional burden on the test environment that must be able to distinguish different test cases.

From the experiences we got while performing the case study we also identified additional problems that had to be considered by **ERLY MARSH**.

- Necessary to also be able to specify and execute manually generated test suites, to ensure inclusion of particularly important test cases. We solved this by implementing support for a user friendly format for specification of individual test cases, part of the formal model of the SUT.
- Large test suites must be supported of all involved parts of the tool, including test suite generation, execution and validation. We solved this by implementing a tool where test suite generation, execution and validation all are integrated parts.
- Presentation of results after executing a generated test suite is essential. We solved this by implementing tool support for clustering detected faults considered similar, and possibility to combine all relevant data for a test case in a single view. For example, from the formal model: trace and parameter values, and from the SUT and test environment: executed trace and log extracts.

Since our evaluation was limited to a single SUT one must be careful with empirical conclusions from the measured results. Nevertheless, our preliminary results were achieved for the type of larger industrial systems that we consider as the target of our research. We therefore have reason to believe that many of our conclusions results will generalize to other similar industrial systems.

9. Other tools for testing ERLANG programs

This chapter presents a small study investigating a few existing tools for testing of ERLANG programs. These tools have different focus (than ERLY MARSH) on how and what to test, thus a fair comparison is hard. One might even argue that they rather complement each other. However, the main purpose for this study was to get a better understanding of the relevance of the results in the previous chapters.

The chapter is organized as follows. Section 9.1 discusses random based testing tools such as QUICKCHECK and PROPER, and Section 9.2 shows the result of using the DIALYZER on A-MLC.

9.1 QUICKCHECK and other random based testing tools

There exists several tools for ERLANG utilizing methods to randomly select test cases, execute those selected, and validate against “properties”. The commercial QUICKCHECK tool [Arts 06, Hughes 10], has lately been followed by other similar open source tools such as TRIQ [Thorup 10] and PROPER [Arvaniti 11].

In property based testing input to a test function is *generated* from a specification of the arguments and output validated against a specification (*property*) of the desired return values. This can be extended to also include systems that have many states. A property model of such a system, expressed as a state machine, can be specified by giving: (1) the initial state, (2) input events (commands) with generators for how data values to input events should be randomly generated, and (3) a set of possible edges, specified by call-back functions on how an input event changes state (`next_state/3`), and (optionally) for conditional generation pre- (`precondition/2`) and post-condition (`postcondition/3`). A property model over A-MLC would need to express all the details on state transitions, guards to input events etc. as the ERLANG/EFMS specification in Section 7.1.1 (ignoring abstraction level). We would also end up in a similar problem as reported on in Section 8.6 with a very inefficient abstract test case selection as input data is generated *before* the precondition check.

Resulting test cases symbolically represents function calls and variable bindings (but *not* parameters to input events) to allow for “shrinking” long test cases to smaller. Optionally it is also possible to specify weights

on edges, where a higher weight implies larger likelihood for an edge to be included in a test case.

The specifications of properties and generators are written directly in **ERLANG** making it possible to operate directly on implementation and use existing type declarations as generators. Also, abstraction/concretization to minimize state space is optional. Thereby the scope of generated test suites normally becomes a little bit different from **ERLY MARSH** that assumes an abstract specification to work efficiently, test cases are symbolically executed to allow for control of test suite generation, and use a constraint solver to find values on depending parameters.

By creating a random test suite with event weights in **ERLANG/EFsm**, see Section 7.2.2, we tried to mimic the creation of a random test suite in e.g., **PROPER**. In both cases, similar models of the expected behavior of the SUT are required. Thus, the effort to create these can be expected to be similar. Although **ERLANG/EFsm** is a version of **ERLANG** dedicated for the purpose of creating such specifications.

In **PROPER** it is only possible to control test suite generation by giving edges different weights. Thus, for what can be expressed in **ERLANG/EFsm**, the possible test suites created with **PROPER** is a subset of the test suites possible to create with **ERLY MARSH**. For this reason, we have not created any test suites with **PROPER**, or any of the other existing property based testing tools in **ERLANG**. A direct comparison with the results using **ERLY MARSH** in Section 8.1 was therefore not possible.

Tools like **PROPER** and **QUICKCHECK** have had some success in the **ERLANG** community. For example, [Boberg 08] report on results from a case study using **QUICKCHECK**. Using this tool, test suites were generated randomly given an abstract state machine and an adaption layer. A conclusion made is that starting model-based testing from early development, significantly increases the number of faults detected during system testing.

9.2 DIALYZER static analysis tool

The **DIALYZER** tool [Lindahl 06] was originally developed at Uppsala University, Sweden and operates directly on **ERLANG** source code (or a debug-compiled object file) without the need to create a formal model of the system or any annotations of the source code. **DIALYZER** uses static analysis techniques, i.e., analysis is performed without actually executing programs, to detect “discrepancies”. Examples of discrepancies detected include type errors and unreachable code such that **DIALYZER** will give warnings if it detects e.g., functions never called and non-matching patterns.

As a model of the system is not required it is much easier to start using **DIALYZER** than e.g., **ERLY MARSH**. **DIALYZER** had not been previously used on the relevant parts of the SUT evaluated in Section 7. It was therefore possible to study the ability of **DIALYZER** to reveal faults not found with manual testing and compare the results with faults found with **ERLY MARSH**.

In order to study the ability of **DIALYZER** to find faults we used the same development release of A-MLC as in Section 7.1. We used version 2.5 of **DIALYZER** and studied the output after analyzing the same **ERLANG** modules on which we measured code coverage in Section 8.4, before any of these modules was updated with bug fixes. All necessary modules for **DIALYZER** to make a complete analysis were included. However, in order to utilize a compatible version of the compiler, for version 2.5 of **DIALYZER**, we also needed to modify the source code of the SUT slightly. One of these changes was related to improvements in the built-in static analysis performed by the compiler, leading to detection of one fault by the compiler.

In total **DIALYZER** needed 21 seconds for a complete analysis. Before any bug fixes 27 warnings were reported, and after all bug fixes were applied 23 warnings remained. The difference on 4 warnings corresponded to 4 faults, exposed as 4 different bug fixes in the case study in Section 7. All of the remaining warnings detected by **DIALYZER** were related to unreachable source code. Comparing with the categorization in Section 8.4 **DIALYZER** found most of the source code reported as unreachable source code in Table 8.6. It can be noted the unreachable source code not reported by **DIALYZER** was of a type requiring manual inspection such as exported but unused interfaces and test functions.

The results can be compared e.g., with the 96 bug fixes detected by All-Paths^{org}. It can also be noted that these 1+4 bug fixes were exposed by all test suites generated with coverage based test case selection except those utilizing the coverage criteria *All target locations*. A conclusion is that static analysis tools, such as **DIALYZER**, are best suited to find faults caused by type and pattern matching problems. The relatively few faults found with the **DIALYZER** tool indicate that the SUT only had few such faults.

With both **ERLY MARSH** and **DIALYZER** it is sometimes difficult to understand what causes the fault and how to fix the bug. The reasons differ though; **ERLY MARSH** gives a complete failing test case, but no detailed info on where to find the fault in the source code. **DIALYZER** gives some detailed information on which function in the source code that fails, but it may be a problem to understand the exact reason why the function fails as the test case is not given. Thus it is hard to decide what test cases were affected by the failure, and the only feedback after a bug has been fixed is that it will then pass the **DIALYZER** test. We can not test

that a bugfix also makes the system behave in a desired way and that the problem was solved in the correct way.

10. Related work

This chapter gives an overview of some related work and is organized as follows. Section 10.1 discuss some relevant generation techniques, tools, and specification languages, Section 10.2 techniques for specifying coverage criteria, and Section 10.3 case studies.

10.1 Test suite generation techniques

From a formal specification of a SUT and a specification of a coverage criterion it is possible to automatically generate a test suite. There exists a wide variety of testing tools, both in academia and in industry, supporting model-based testing and test suite generation. The capabilities of these approaches differ because of different priorities concerning usability, expressiveness, and efficiency of test suite generation. In general they are therefore hard to compare, although there exist comparisons, e.g., [Goga 01, Sinha 06, Utting 12, Shafique 15]. Here we will only briefly outline test suite generation tools and specification languages supporting similar technologies as presented in this thesis, i.e., black-box, coverage-based and model-based test suite generation.

For a user of model-based testing, the most effort-requiring, labor-intensive task is the process of creating the formal model. There are many aspects of this, e.g., a specification language should be easy to use and able to express desired properties in desired way. Basic components of a specification language includes e.g., constructs for specifying components such as types and functions, structuring mechanisms for building large specifications, and a way of relating specifications to implementations. In each case there are various alternatives to choose from, but no single best combination because this depends on e.g., purpose of test specification and type of SUT. Existing specification languages used for coverage-based model-based test suite generation includes e.g., Lotos [Bolognesi 89], SDL [ITU-T 99b] and UML Statecharts [Gnesi 02, OMG 03]. In Section 2.3 we suggested the use of `ERLANG/EFsm` for the specification of state machines. In contrast to the above list of specification languages, `ERLANG/EFsm` has its roots in a functional language with an ability to express functional requirements in a concise way.

One argument introducing yet another specification language is our belief that all parties involved in the software development process benefit

using a common language. Similar arguing can also be found in motivations using the model-based testing tool Spec Explorer [Veanes 08] and the open source variant NModel [Jacky 08] that use the C# syntax (or AsmL) as specification language for testing implementations in e.g., C#. For Java implementations also the Java Modeling Language (JML) [Leavens 99] allows for coverage-based test suite generation of Java implementations by the tool Korat [Boyapati 02]. Although this approach is different in requiring augmenting Java methods in the implementation with pre-conditions and post-conditions, i.e., white-box testing.

A popular approach in model-based testing is to use a model checker to find a counterexample which can be transformed into a test case. State space is searched from an initial state by forward exploration until a violation of a property is found. The counterexample (i.e., test case) is simply the explored path back to the initial state. See e.g., [Fraser 09] for an overview on test suite generation with model checkers. In the following we distinguish between *Explicit model checking*, requiring an explicitly represented state space, and *Symbolic model checking* [Burch 92, Lin 96] requires sets of states to be handled symbolically.

In *Explicit model checking* due to the state space being explicit, finite-state models are assumed. An effect of this is that data variables in the model must often be assigned specific values in order to generate test cases. An early example is Autolink [Koch 98] that was used (part of Tau/Telelogic, now Rational Tau [IBM 11]) for generating test suites from SDL and UML specifications. Other early examples are (from Lotos or SDL specifications) the TGV tool set [Jard 05] and (from Lotos, PROMELA or FSP specifications) the TorX tool set [Vries 00, Tretmans 02], later rewritten into JTorX [Belinfante 10]. Both TGV and TorX are based on IOLTS (Input Output Labeled Transition System) and use the **io**co conformance relation [Tretmans 96] which characterizes the relation between the formal model and the SUT. Test suites are generated by first creating a synchronous product between a test purpose and a specification. The resulting EFSM then represents a set of test cases satisfying the test purpose, a set of test cases failing according to the **io**co relation, and possibly a set of correct test cases but only partially covered by the test purpose.

PROMELA (Process Meta Language) [Holzmann 03] is a language originally developed as the input language to the tool SPIN. The SPIN tool is a model checker that supports verification of properties expressed in Linear Temporal Logic (LTL), but was also early used in model-based testing, e.g., on-the-fly testing in [Fernandez 96]. More lately it has also been used in e.g., [D'Souza 03] for checking consistency between a SDL specification and test purposes expressed as MSC Specifications.

Symbolic model checking is by definition not limited to finite state spaces. To represent the sets of states, and function relations on these

states, (typically) BDDs are used. Thus, in practice the selected technique to represent the sets of states (e.g., BDDs) enforce a finite state space.

There exists several approaches using the model checker SMV for example, [Gargantini 01] (from ASM specifications) and [Rayadurgam 01] (from RSML specifications) as well as its derivative NuSMV [Kadono 09] (from Statechart specifications) and [George 12] (from RSML specifications). In [Friedman 02] the Mur Φ model checker [Ip 99] is used to generate test suites, using coverage criteria specified in terms of projections in the GOTCHA tool. The STG tool [Clarke 02, Frantzen 05] use an extension of IOLTS, known as IOSTS, supporting guards and state variables. Based on the theory behind the TGV and TorX tools, but with symbolic transitions on state graphs, test suites are generated by first creating a synchronous product between a test purpose and a specification. The resulting IOSTS then contains a symbolic representation of the set of test cases satisfying the test purpose. Also the Agatha tool [Rapin 03] use a similar extension of IOLTS, known as EIOLTS (Extended IOLTS).

More recent tools include the cover tool [Hessel 07, Hessel 08] using observer automata on timed automata to specify coverage criteria. Test suites are generated using Uppaal [Larsen 97] (from Timed Automata specifications), an explicit model checker with symbolic handling of clocks. Commercially available tools include e.g., Conformiq Designer [Huima 07, Sving 10] that generate test suites using symbolic execution. Several specification languages are supported, e.g., UML Statecharts, and several different coverage criteria can be used. Resulting test suites can be represented in e.g., TTCN-3 [TTC 03], where a test suite is formulated as a (parameterized) tree when a SUT respond in different ways to the same stimuli.

Theorem provers use inference rules and axioms to automatically (if possible) derive proofs of properties. Theorem provers are For model-based testing approaches the SUT is modeled by a set of logical expressions (predicates) specifying the SUT's behavior For example, [Brucker 13] use the higher-order logic theorem prover system Isabelle to encode state machines and test derivation strategies. Proofs of Similar to Symbolic model checking tools, theorem provers are not limited to finite state spaces, as long as the derived proofs can use inference rules with symbolic parameters.

10.2 Coverage criteria and Test purposes

The idea to be able to specify (almost) arbitrary coverage criteria using a dedicated formalism is not new. The expressiveness in these formalisms vary: some formalisms target more specific test purposes, others allow

specification of more generic coverage criteria, see Section 1.5.3. The distinction between test purposes and coverage criteria is not sharp. If we regard a test purpose as a specific part of functionality to be covered by a test case, then a coverage criterion can be regarded as a (sometimes huge) collection of test purposes (coverage items in the terminology of Section 3.1).

There are several approaches to specifying coverage criteria in terms of individual test purposes. In [Mandrioli 95] a linear time temporal logic, TRIO, is used for specification of a model of the SUT and coverage criteria. In [Gargantini 03] several coverage criteria is specified using so-called “test predicates”, each of which specifies a particular testing goal, e.g., whether a particular statement has been executed. They thereafter use the SPIN model checker to search for an execution which meets this testing goal. In a slightly earlier work [Gargantini 01] the branching time temporal logic, CTL, is used for specification of coverage criteria, something they share with [Rayadurgam 01]. Also [Hong 02, Hong 03] describe how flow-based coverage criteria can be expressed in CTL. A particular coverage item is expressed in CTL, and a model checker generates a trace which covers the coverage item. In particular they also give several examples of how more complicated data-flow coverage criteria, as Def-Use, can be accomplished.

In several approaches, automata are used to specify individual test purposes. In [Jard 05], and also [Tretmans 02], finite automata is used, with accepting and rejecting states to express test purposes. A problem with pure temporal logic or finite automata is that parameterized scenarios are hard to describe. In [Rusu 00] this is addressed by using (not necessarily finite-state) symbolic automata to represent test purposes. This allows test purposes to consider data parameters in the generation of test cases. Of course, the test case generation must now deal with symbolic expressions over the data parameters. However, they do not support the generation of a *set* of test cases, indexed by some parameters, as in our use of parameterized observers.

Our use of observer automata for generation of test suites are also related to the work of [Friedman 02]. In their approach, a coverage criterion is specified by means of a projection of the state space of an EFSM onto a subset of its variables. For each combination of values of these variables, a corresponding test suite should contain a test case which visits that combination. The tool used for creating test suites, GOTCHA, is capable of handling projections onto any of the syntactical components which are part of an EFSM.

The main novelty of our approach lies in the ability to specify test suites by means of coverage criteria that are parameterized on arbitrary syntactical components. For example, the possibility to express a coverage criterion, such as All-Locs, by means of quantification over the meta-

variable “locations” is not present in other approaches. Instead, coverage of each location has to be expressed explicitly, e.g., by means of a separate formula for each location. This means that test suite generation must employ repeated searches of the state space.

The idea of an observing automata watching the behavior of a state graph, can also be found to be used for other purposes than testing. This idea goes back to the “automata-theoretic approach” to model checking [Vardi 86]. Uses of this idea include [Alfaro 01], who observe “compatible” sequences events on an interface between two components to find environments on which they are compatible. Automata are also often used to specify temporal properties. For example, [Beyer 04, Beyer 13] monitors execution of C programs in the BLAST tool to verify temporal properties of traces. Later BLAST has also been used for test suite generation [Beyer 13] (white-box testing directly on the implementation) by the use of a model checker in a similar fashion as described in Section 10.1.

In Section 8.3 we noted that choosing the original or normalized base model had an effect of the outcome. These results are also supported by [Heimdahl 04] that argue common coverage criteria are inadequate because they do not consider the logic of the model. I.e., knowledge of the model required to use a proper search strategy and avoid testing parts of the specification most easily accessible.

We did not try to further compare the coverage criteria and test suites against each other as it would have required some normalization of the results. It can be noted approaches exists in the literature using a reliability growth model, e.g., [Malaiya 02], where the reliability of the SUT is measured in e.g., number of failures detected. Over time, during development, the number of failures detected can be expected to decrease and reliability increase. Comparing how reliability increase with different testing strategies (e.g., coverage criteria) over time then make it possible compare them.

Further comparisons between different coverage criteria is needed in order to better understand when to use a particular coverage criteria and how to generate it most effectively. As also stated in [Bertolino 07], the existing number of coverage criteria is so large that it is a real challenge to understand how to make a justified choice, or understand how several coverage criteria can be combined most efficiently. The decision on how test suites should be generated must consider not only the functional requirements, but also the time to generate and execute test suites.

10.3 Case studies

There exists many case studies of model-based testing in the literature, see Section 10.3, but to our knowledge none with a comparable size and scope

evaluating coverage based testing. The large amount of time spent on this case study is also the main motivation on why only a single evaluation was performed.

In the literature there exists many case studies on different types of software testing. On model-based testing most of such case studies are either not focused on evaluating coverage based testing, small, using non-industrial development methods (e.g., open source), or on SUTs populated with generated faults e.g., [Fernandez 97, Bouquet 03, Tretmans 03, Mäkinen 07, Santos-Neto 08]. Thus, they may not reveal the same results as testing industrial size products with real faults. To our knowledge, there exists no published results of industrial case studies with directly comparable results. Here we will only briefly outline case studies found most relevant in the literature.

Most closely related studies include [Heimdahl 04, George 05, Staats 12], which differ in that faults are injected by mutation. In [Heimdahl 04] coverage-based test suites are generated using a different technique than in our tool. This results in an abundance of very short test cases, with a modest ability to detect faults. They explain the poor fault detection ability by observing that the structure of the system does not well match the coverage criterion, and that criteria based on decision coverage may be deficient if they do not force the test suite to exercise relevant data flows. The findings by Heimdahl et al. contrast with the findings from our case study, in which the coverage-based test suites perform comparatively well. Contrasting with [Heimdahl 04], we note that our test generation technique, using observers, does not have an excessive bias towards short test cases, and that the data flow coverage criterion of Def-Use^{org} gives excellent fault detection power. Poor performance of only coverage-based test suites was again observed in [Staats 12]. Reading this, one possible explanation is that their counterexample-based technique for test suite generation creates some bias. It could be the case that our observer-based technique avoids such bias, but deeper investigations are needed before this can be confirmed.

In [Pretschner 05] model-based testing is compared with manual testing in terms of quality and cost. Generated test suites were created by randomly selecting test cases from a model of an automotive infotainment system. The largest generated test suite included 1000 test cases. A finding of the study was that the generated test suites found significantly more requirements faults compared to manual testing while the number of detected programming faults was approximately equal. Also, no correlation between severity of errors and types of test cases were found.

Spec Explorer is a popular Model-based testing tool by Microsoft that is used in several case studies. As reported in [Grieskamp 11] Spec Explorer is also used extensively internally at Microsoft. From the preliminary results in this study 125 protocols with an investment of 50 person

years has resulted in a 42% productivity gain when compared to traditional (i.e., manual) testing. In another industrial case study [Sarma 10] by Siemens Spec Explorer [Veanes 08] is compared with Qtronic [Huima 07]. A general conclusion is that these tools deserve to be considered in industrial projects. Nevertheless, a number of shortcomings are identified in both tools including:

- Integration with different types of testing, e.g., performance testing and unit testing (see also Section 1.1).
- More possibilities to optimize test suites e.g., by prioritization of test test cases or design of better coverage criteria.
- Support of round-trip engineering, i.e., faults detected should be easy to map to the corresponding specification (e.g., by highlighting relevant part).

Five different case studies are reported in [Weißleder 10], with test suites ranging from 10 to less than 1000 test cases. All these case studies use a specification in UML and mutated implementations to compare 5 different coverage criteria. A conclusion is that a combination of coverage criteria often performs best.

In [Weiglhofer 09] a mutation of the specification is used in two case studies utilizing fault-based testing (see 1.5.3). In these case studies fault-based testing is compared with random testing and testing based on hand-crafted test purposes. A conclusion is that fault-based testing finds faults not found by the other methods. However, unclear if the faults only found by the fault-based testing could have been found as effectively with better coverage criteria.

Out of the many existing case studies on testing, not utilizing an abstract model of system to generate test suites from (i.e., not model-based testing), we also like to mention two case studies. In [Janhunen 11] coverage based testing is compared with random testing on declarative programs (the Answer set programming language Gringo). The results indicate that random testing is quite ineffective for some benchmarks, while coverage based techniques catch faults with a more consistent rate. However, even if coverage based testing gives a clear advantage over random testing for some SUTs, for other SUTs one approach can not be said to be better than the other.

As part of the European project EvoTest two larger industrial case studies were performed [Vos 12]. Although these studies use a different test methodology (Evolutionary Testing based on Artificial Intelligence), i.e., not model-based testing, the results indicate that a computer assisted effort spent on test suite generation pays off compared to manual and random testing.

11. Conclusions

This chapter concludes the thesis by summarizing and discussing the results presented in this thesis. It is organized as follows. Section 11.1 gives a summary of the achievements of the thesis. In the following sections we discuss the research contributions in more depth. Section 11.2 on our `ERLANG` based modeling languages, Section 11.3 on the flexible observers, Section 11.4 on the symbolic test suite generation, Section 11.5 on how we concretize abstract test cases in the case study, Section 11.6 on our tool for support of model-based testing, and Section 11.7 evaluate different test suite generation strategies. In Section 11.8 we further discusses the work in this thesis, and Section 11.9 presents some ideas for future work.

11.1 Summary

This thesis is about model-based protocol testing of an implementation of a SUT in an `ERLANG` environment. For specification of models we have introduced a specification language, `ERLANG/EFsm`, based on the functional language `ERLANG` in Chapter 2. We have based `ERLANG/EFsm` on `ERLANG` and combined this syntax and semantics with explicit notation for expressing abstract state machines. Examples of additions to `ERLANG` include additional constructs to handle locations and global state variables. The semantics for `ERLANG/EFsm` was given as a big-step structural operational semantics in the form of transition rules between structural states. We have also given a symbolic semantics with transition rules for symbolic execution where symbolic parameters are not evaluated. The close relationship with `ERLANG` was also advantageous for the straightforward translation of an `ERLANG/EFsm` specification into an executable `ERLANG` module.

We have introduced *observers* (automata) as a tool for flexible specification of coverage criteria, in Chapter 3. An observer is a state machine that monitor `EFsm` runs by utilizing observer predicates as conditions for a test case to be included in a test suite. The user-definable observer predicates control execution of the observer by utilizing match variables, the interface between the observer and the `EFsm`. The syntax for specifying observers, `ERLANG/OBS`, has many similarities with `ERLANG/EFsm` but is more limited since observers, in general, are much simpler in structure. We have also developed a graphical notation for specification of observers.

The semantics for **ERLANG/OBS** was based on the semantics for **ERLANG/EFSM**, in the form of transition rules between structural states. We also gave transition rules for symbolic execution of observers. In Chapter 4 we gave several examples on the flexibility of observers by giving several examples on coverage criteria expressed as observers in **ERLANG/OBS**.

We have presented our technique for generating test suites from **ERLANG/EFSM** models extended with observers in Chapter 5. Here we gave a symbolic state space exploration algorithm for test suite generation. We noted that for handling large test suites, an efficient representation of guards and path conditions is crucial. To handle complex observers we identified the need to efficiently represent large sets of observer states.

We have given an overview of **ERLY MARSH** in Chapter 6. This is a tool we have developed for model-based test suite generation and test suite execution. An ambition with this tool was to cover all aspects required for industrial usage of model-based testing. Thus, in addition to test suite generation, we also integrates possibilities to validate the **ERLANG/EFSM** specification, concretize and execute generated abstract test suites, and generate reports on the outcome of executing both individual test cases and test suites.

The setup for our case study, where we use our techniques for model-based test suite generation, was presented in Chapter 7. The results of using **ERLY MARSH** on this case study was presented in Chapter 8. In Chapter 9 we compared these results with other tools for testing and verifying **ERLANG** programs.

11.2 Modeling Language

One of the main motivations for developing a new modeling language was to make it easier for **ERLANG** developers to make formal models. In **ERLANG/EFSM** we combine the syntax and semantics **ERLANG** with additional notation for more explicit expression of abstract state machines. The result is a language specifically targeted for specification of protocol modules. Our experience from using **ERLANG/EFSM**, in particular when creating the A-MLC model, is that the particular features of **ERLANG** that are retained in **ERLANG/EFSM** make modeling of protocol modules quite convenient. This is not surprising, since **ERLANG** was originally developed for programming communication protocols. A prominent feature of **ERLANG**, such as pattern matching, allows to make compact representations of specifications, both of parameters in input events, and of any expression triggered by the occurrence of an input event. For example, complex data structures, such as records, in **ERLANG/EFSM** can be decomposed in a convenient way. Another feature in **ERLANG** is the possibility to structure the definitions of functions decomposed into several function clauses. In

ERLANG/EFMS we can use this to specify different input event types into different function clauses, to get more intuitive mappings to edges in an **EFM**S.

There exists other modeling languages that have a well-developed tool support (e.g., Spec Explorer). But to learn mastering a new language and tool is a hurdle for anyone not previously familiar. Therefore, one can argue that for **ERLANG** developers, already well accustomed with syntax and editors to develop **ERLANG** programs, it is a small step to learn to create models in **ERLANG/EFM**S and observers in **ERLANG/OBS**.

The close relationship with **ERLANG** also had the implication that we could use the existing **ERLANG** parser when parsing models in **ERLANG/EFM**S and observers in **ERLANG/OBS**. Thus, as **ERLY MARSH** is written in **ERLANG**, development of this tool could be developed faster and more reliably. Naturally, implementing **ERLY MARSH** in **ERLANG** also makes it easier to implement an interface to a SUT written in **ERLANG**.

11.3 Specifying Test Case Selection

There exist many types of SUTs, and each can be specified in many ways. Further, each specification of a SUT can be utilized for model-based testing in many ways. This calls for a need of flexible specification of coverage criteria. Using the observer automata, a selection of such coverage criteria was given in Chapter 4. In fact, with the limitations implied by what can be expressed in **ERLANG/EFM**S, we have been able to specify all existing coverage criteria we have found in the literature. Further examples of the flexibility of the observer automata is given in Section 5.5 where we show that only slight modifications are needed in order to use observer automata as a filter (for inclusion of all test cases that share a certain property) or for property validation.

There are costs incurred by using more complex coverage criteria, and in the flexibility of using observers to implement the coverage criteria. How high these costs are depends on the complexity of the coverage criteria and the implementation of the tool used. For an implementation using observers, a main cost is related to the book-keeping required for each additional observer location in the observer. When we need a compact representation and utilize a lot of book-keeping, observer locations might be most efficiently represented by bitvectors. In other situations some other choice be a better alternative. In the case study we used bitvectors to represent all observer locations and showed, in Section 8.5 that the additional time required for a more complex observer (Def-Use) to generate a test suite can be significant.

11.4 Efficient Generation of Test Suites

Before test suite generation can begin we need to create a model. To validate a created model without executing test cases against a SUT, **ERLY MARSH** offers:

- model checking by using a slight modification of the symbolic state space exploration algorithm for test suite generation and observer automata, see Section 5.5.2, and
- the possibility to use the **ERLY MARSH** integrated simulator and animate the specification, see Section 6.1.

In this thesis we have concentrated on functional system testing where the state space of the implementation of the SUT is large and we therefore based our test suite generation on an abstract model of the system. To efficiently generate test suites we used a combination of several existing approaches, in particular

- *Abstraction macros* to fine-tune the abstraction level in the specification. This allowed us to express details in the specification more clearly and use more natural domains on parameters. Examples include e.g., sets of status codes from standards that, with an abstract view, should be treated in the same way. Abstraction macros let us keep all the different concrete values in the specification, without increasing the state space. In the case study we used 9 abstraction macros to increase readability of the specification and limit the state space.
- *Partial evaluation* to limit the size of symbolic expressions on each edge in the EFSM. By evaluating all expressions (e.g., case and if expressions) in the transition clauses in the specification as much as possible we were able to greatly simplify expressions. Thus, the resulting edge clauses had much simpler expressions which was beneficial for test suite generation.
- *NDD:s* to efficiently represent and store guards and path conditions. In the case study it was clear that due to the large state space and often complex guards it was crucial to have an efficient representation of guards and path conditions. But it was only after evaluating several alternative representations that the NDD:s representation were chosen.
- *Symbolic execution* to efficiently generate larger test suites. With an efficient symbolic representation of the guard of each edge it was a natural choice to also symbolically execute all other expressions in the edge during test suite generation. This limited the problem with state space explosion. Symbolic execution was particularly successful in our case study, due to the lack of complicated constraints to solve when selecting abstract test cases, after symbolically executing the specification.

In the case study we partially evaluated all transition clauses in the EFSM improving speed in test suite generation and without losing any significant details. The resulting state space, after using all of the above mentioned approaches, in the abstract model used for test suite generation was covering more than 2^{38} abstract test cases. Without a symbolic approach it would not have been possible for us to generate such a large number of abstract test cases off-line.

11.5 Concretization of Generated Test Cases

For any **ERLANG/EFMS** specification, which represents an abstraction of a SUT there is a need to concretize any input to the SUT (abstract any output). In **ERLY MARSH** this is solved by the use of functions in a call-back module to handle the concretization and abstraction.

The created A-MLC model in **ERLANG/EFMS** had a rather low level of abstraction. An implication of this was that the concretization/abstraction functions were often simple mappings, because corresponding input/output expressions and configuration access functions in the formal model could easily be identified with real elements in the implementation. For example, an input expression such as *slir(MS, MlpAge)* had straightforward mappings with elements and attributes in the XML document carrying the concrete *slir* event as accepted by the SUT. In the case study there was a number of such often complex data types/large domains in the implementation that were mapped against simpler data types/smaller domains in the formal model.

11.6 Efficient Test Execution and Verification

When executing test cases concurrently there is a need to separate different test cases. In **ERLY MARSH** this is solved during concretization by using parameters in concrete elements to carry test identifiers. Naturally, this requires the existence of such available parameters. In the case study a considerable amount of time was saved by the possibility in **ERLY MARSH** to execute test cases concurrently.

Executed test cases must conform to the formal model after applying any concretization/abstraction functions, and results must be presented so that they can be understood by a user. **ERLY MARSH** is able to generate a report of each test case executed with information on which part of the test case that conformed with the formal model. SUT dependent data, e.g., logs, counters and raised alarms, may optionally be added to such generated reports. Further, **ERLY MARSH** generates a summary of all test case results after a test suite has been executed. The generation of these reports were highly essential when performing the case study.

11.7 Evaluation of Different Test Generation Strategies

In the evaluation, we generated test suites satisfying different coverage criteria. For comparison, we also included a number of randomly generated test suites of similar sizes, and a hand-crafted manual test suite.

Ideally we would be able to measure the quality of generated test suites. An important distinction in our case study is the use of the measure *bug fixes* (see Section 7.3.2). That is, we used existing, real faults in the SUT when comparing different test suite generation strategies. We argue that a measure of real faults has benefits, in particular as in the case study, when studying software where obvious faults are fixed. Interestingly, examples from the case study includes functional faults that were caused by inconsistent behavior and over utilization of external interfaces. These are examples of faults, with potentially great impact, that can be found only with test cases in which actual values of input and output are provided. A consistent description of test cases was further enforced by the formally expressed state machine in `ERLANG/EFM`. The relatively few faults found with the `DIALYZER` tool in Section 9.2 also indicate that the kind of faults found with `ERLY MARSH` are hard to find with other approaches.

Test suite generation depends on a combination of such diverse aspects as model of the SUT, time frame, and type of testing. We did therefore not try to designate a coverage criterion as the “best choice”, and note that what is used as a measurement may also affect the results.

A much debated topic in software testing is whether it is most efficient with a partition based technique to select test cases to be part of a test suite, or if they could equally well be selected completely random e.g., [BJK 04, Janhunen 11]. Our experience with random test suites from Section 8.1 showed no clear difference in power between coverage-based test suites and randomly generated test suites of the same size, as measured by the number of detected faults and source code coverage. As the size of test suites increased, the differences between different test suites decreased. It should then be remembered that for the random test suites, the results very much depended on information provided by the A-MLC specification. Also the fact that, for this particular SUT, we got good results after executing many short test cases favoured the random test suites with such characteristics, i.e., those using event weights. See also summaries of findings on case studies comparing random testing in Section 10.3 and discussion in Section 8.6.

11.8 Discussion

Model-based testing is more than just generating an abstract test suite from a formal model. To be useful in an industrial setting we must consider the complete picture, from creating and validating the formal spec-

ification to verifying results of executing a test suite. Of particular interest was if model-based testing would deliver results to be useful for an industrial sized systems. This thesis contributes to this area by the case study.

The case study started out with the ambition to handle the problem on how to handle testing of software in a small company. A prerequisite was therefore that there only existed a limited amount of resources. Identified needs included functional testing, but also performance testing, and testing required with software problems reported. This required a flexible solution in order to reuse as much as possible between the different types of testing needed. Given an existing detailed functional specification of the SUT, the use of an integrated tool supporting model-based testing therefore seemed a natural solution.

When a software problem is reported it might be interesting to generate test suites based on available knowledge of the problem. Thus, be able to reproduce the problem in a controlled environment. Using observers we can then generate test suites covering the fraction of the SUT where the software problem occurs. Further, whenever a fault is detected and a bug fix has been applied we have gained some knowledge about the fault. Again we then might want to use an observers to generate test suites covering not only (previously) failing test cases, but also test cases dependent on the bug fix, to verify the bug fix.

A small company is often characterized by an agile workflow with short distances between the different roles. Development of a formal model simultaneously with the implementation can therefore be beneficial also from a communication perspective between people. Later, when maintaining the system, the formal model acts both as a document describing the system and as source to generate regression test suites from (using observers). It is therefore our belief that model-based testing has strong merits to be used for testing implementations of industrial size products on a functional level, in particular using **ERLY MARSH** for systems implemented in **ERLANG**.

11.9 Future Work

We will in the following briefly discuss some additional directions for future work, including e.g., how the framework can be extended to make evaluations of different systems and aspects of testing possible. But we can also note that the possibly most urgently needed future work are additional evaluations on how generated test suites behave on industrial systems. In particular, to contribute to the important question on how to aid in the determination on how to select better test suites to generate for a particular occasion and/or SUT.

11.9.1 Creating specifications

There exists additional features part of `ERLANG`, not part of `ERLANG/EFsm`, that would potentially be useful in a specification language. An important concept in `ERLANG` is the use of communicating processes, that may send messages between each other using mailboxes. An addition of processes to `ERLANG/EFsm` is possible but as the purpose is specification of systems, other communication strategies than mailboxes are possible. However, the need of support for “true” concurrency in `ERLANG/EFsm` is not clear. For example, one may argue that partitioning into communicating processes is a design decision and should not be part of the specification [Jantsch 00]. Assuming the support for concurrency in `ERLANG` can be trusted on also limits the need to generate test suites of implementations in `ERLANG`, verifying an application handles concurrency properly.

Missing is also the possibility to structure larger specifications into smaller modules. Several approaches are possible. It may, for example, be desirable to introduce hierarchically nested states as in UML Statecharts. Nested states can then be specified in separate modules. There exists other examples of possible extensions of `ERLANG/EFsm`, e.g., improved support for cyclic behaviours and non-deterministic behaviours. But it may be noted that addition of new features would possibly imply that the current big-step operational semantics of `ERLANG/EFsm` could not be extended on.

The specification language used to express a test suite is restricted by how guards and path conditions are represented. In the evaluation in Section 7 we used NDD:s to represent path conditions as it was a suitable choice for this particular system. For other specifications of systems, an alternative representation of guards and path conditions might be more appropriate. Thus, to optimally representing test suites it is relevant to explore some of the large number of efficient representations of different subsets of expressions found in the literature.

11.9.2 Generating test suites

The current framework is limited to support test suite generation on finite, deterministic specifications with a set of stop locations only. All of these are components that in some cases may be considered more adequate to generalize. How to do that is considered future work, but we do not expect major updates to be required in our test suite generation algorithm and tool.

Together with an efficient representation of test cases, the search strategy maybe has the most significant impact on the performance of the test suite generation algorithm. In this thesis we did only investigate depth-first search. One interesting idea of an alternative search strategy would

be to create a concurrent algorithm where multiple branches of the search are executed concurrently. For example, using the technique described in [Staats 10] where different evaluations, of frequently occurring symbolic parameters in edge clauses, are separated into different concurrently executed branches. A concurrent algorithm also requires the use of a global data structure that can be updated by each concurrently executed branch.

Observer predicates are user definable and we allow match functions, used to help define observer predicates, to also include additional `ERLANG` expressions. However, the current version of `ERLY MARSH` only supports a set of predefined observer predicates and conditioned observer states $\langle \iota(\bar{e}), H \rangle$ that do not have any symbolic parameters, and where H is `true`. We did also not yet include a constraint solver simply because it was not needed for the A-MLC `ERLANG/EFM` specification in the evaluation. A consequence is that `ERLY MARSH` currently can not generate all test suites that is possible to express with `ERLANG/EFM`.

Abstract test case selection

Further possible improvements of the abstract test case selections include:

- For each symbolic test case, let the number of abstract test cases, covered by the symbolic test case, decide if more than one test case should be selected. Thus, randomly selecting multiple abstract test cases from symbolic test cases with many possible abstract test cases.
- For each symbolic test case, use a combination strategy, such as all pairs of parameter values see Section 1.5.3, to decide if more than one test case should be selected. Thus, selecting multiple abstract test cases from symbolic test cases with many possible parameter value combinations.

References

- [Aichernig 12] AICHERNIG, BERNHARD K. 2012. Model-Based Mutation Testing: Theory and Application. Habilitation thesis, Graz University of Technology, Austria.
- [Aichernig 09] AICHERNIG, BERNHARD K. AND JIFENG, HE. 2009. Mutation testing in UTP. *Formal Aspects of Computing, Volume 21*, no. 1-2, pages 33–64.
- [Alfaro 01] ALFARO, LUCA DE AND HENZINGER, THOMAS A. 2001. Interface automata. In *Proc. ESEC/FSE*. ACM, pages 109–120.
- [Ammann 08] AMMANN, PAUL AND OFFUTT, JEFF. 2008. *Introduction to Software Testing*, 1 edition. Cambridge University Press, New York, NY, USA.
- [Ammann 03] AMMANN, PAUL, OFFUTT, JEFF, AND HUANG, HONG. 2003. Coverage Criteria for Logical Expressions. In *Proc. 14th Int. Symposium on Software Reliability Engineering (ISSRE '03)*. IEEE Computer Society, pages 99–107.
- [Ammons 03] AMMONS, GLENN, MANDELIN, DAVID, BODÍK, RASTISLAV, AND LARUS, JAMES R. 2003. Debugging temporal specifications with concept analysis. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, pages 182–195.
- [Anand 13] ANAND, SASWAT, BURKE, EDMUND K, CHEN, TSONG YUEH, CLARK, JOHN, COHEN, MYRA B, GRIESKAMP, WOLFGANG, HARMAN, MARK, HARROLD, MARY JEAN, MCMINN, PHIL, ET AL. 2013. An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software, Volume 86*, no. 8, pages 1978–2001.
- [Arts 06] ARTS, T., HUGHES, J., JOHANSSON, J., AND WIGER, U. 2006. Testing Telecoms Software with Quviq QuickCheck. In *Proc. 2006 ACM SIGPLAN workshop on Erlang*. ACM, Portland, Oregon, USA, pages 2–10.
- [Arvaniti 11] ARVANITI, EIRINI. 2011. Automated Random Model-Based Testing of Stateful Systems. Diploma thesis, National Technical University of Athens.
- [Asarin 97] ASARIN, E., BOZGA, M., KERBRAT, A., MALER, O., PNUELI, M., AND RASSE, A. 1997. Data Structures for the Verification of Timed Automata. In *Int. Workshop Hybrid and Real-Time Systems (HART)*, Volume 1201 of *Lecture Notes in Computer Science*. Springer, pages 346–360.
- [Barklund 99] BARKLUND, J. AND VIRDING, R. 1999. ERLANG 4.7.3 Reference Manual.

- [Beizer 90] BEIZER, B. 1990. *Software Testing Techniques*. Van Nostrand Reinhold, New York.
- [Belinfante 10] BELINFANTE, AXEL. 2010. JTorX: A Tool for On-line Model-driven Test Derivation and Execution. In *Proc. Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Volume 6015 of *Lecture Notes in Computer Science*. Springer, pages 266–270.
- [Bertolino 07] BERTOLINO, ANTONIA. 2007. Software Testing Research: Achievements, Challenges, Dreams. In *Int. Conf. on Software Engineering, (ICSE), Workshop on the Future of Software Engineering, (FOSE)*. IEEE Computer Society, pages 85–103.
- [Beyer 04] BEYER, DIRK, CHLIPALA, ADAM J., HENZINGER, THOMAS A., JHALA, RANJIT, AND MAJUMDAR, RUPAK. 2004. The Blast query language for software verification. In *Proc. 11th Int. Symp. Static Analysis (SAS)*, Volume 3148 of *Lecture Notes in Computer Science*. Springer, pages 2–18.
- [Beyer 13] BEYER, DIRK, HOLZER, ANDREAS, TAUTSCHNIG, MICHAEL, AND VEITH, HELMUT. 2013. Information Reuse for Multi-goal Reachability Analyses. In *Proc. 22nd European Symp. on Programming (ESOP)*. Volume 7792 of *Lecture Notes in Computer Science*. Springer, pages 472–491.
- [Binder 99] BINDER, R. 1999. *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Addison Wesley.
- [Bishop 05] BISHOP, STEVE, FAIRBAIRN, MATTHEW, NORRISH, MICHAEL, SEWELL, PETER, SMITH, MICHAEL, AND WANSBROUGH, KEITH. 2005. Rigorous specification and conformance testing techniques for network protocols, as applied to TCP, UDP, and sockets. In *Proc. ACM SIGCOMM 2005 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*. ACM, pages 265–276.
- [Boberg 08] BOBERG, JONAS. 2008. Early fault detection with model-based testing. In *Proc. 7th ACM SIGPLAN workshop on Erlang, ERLANG '08*. ACM, New York, NY, USA, pages 9–20.
- [Bolognesi 89] BOLOGNESI, T. AND BRINKSMA, E. 1989. Introduction to the ISO Specification Language LOTOS. *Computer Networks*, Volume 14, no. 1 (Jan.), pages 25–59.
- [Bouquet 03] BOUQUET, F. AND LEGEARD, B. 2003. Reification of Executable Test Scripts in Formal Specification-Based Test Generation: The Java Card Transaction Mechanism Case Study. In *Proc. Int. Symp. on Formal Methods Europe (FME)*, Volume 2805 of *Lecture Notes in Computer Science*. Springer-Verlag, pages 778–795.
- [Boyapati 02] BOYAPATI, RASEKHAR, KHURSHID, SARFRAZ, AND MARINOV, DARKO. 2002. Korat: Automated testing based on Java predicates. In *Proc. Int. Symposium on Software Testing and Analysis (ISSTA)*. ACM Press, pages 123–133.
- [Briand 07] BRIAND, LIONEL C. 2007. A Critical Analysis of Empirical Research in Software Testing. In *First Int. Symposium on Empirical*

- Software Engineering and Measurement*. IEEE Computer Society Press, pages 376–387.
- [BJK 04] BROJ, M., JONSSON, B., KATOEN, J.-P., LEUCKER, M., AND PRETSCHNER, A., EDITORS. 2004. *Model-Based Testing of Reactive Systems*. Volume 3472 of *Lecture Notes in Computer Science*. Springer-Verlag.
- [Brucker 13] BRUCKER, ACHIM D. AND WOLFF, BURKHART. 2013. On theorem prover-based testing. *Formal Aspects of Computing*, Volume 25, no. 5, pages 683–721.
- [Bryant 86] BRYANT, R.E. 1986. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Trans. on Computers*, Volume C-35, no. 8 (Aug.), pages 677–691.
- [Burch 92] BURCH, J.R., CLARKE, E.M., MCMILLAN, K.L., DILL, D.L., AND HWANG, L. J. 1992. Symbolic Model Checking: 10^{20} States and Beyond. *Information and Computation*, Volume 98, no. 2 (June), pages 142–170.
- [Chilenski 94] CHILENSKI, JOHN JOSEPH AND MILLER, STEVEN P. 1994. Applicability of Modified Condition/Decision coverage to software testing. *Software Engineering Journal*, Volume 9, no. 5 (September), pages 193–200.
- [Ciupa 07] CIUPA, ILINCA, LEITNER, ANDREAS, ORIOL, MANUEL, AND MEYER, BERTRAND. 2007. Experimental assessment of random testing for object-oriented software. In *Proc. Int. Symposium on Software Testing and Analysis (ISSTA)*. ACM, New York, NY, USA, pages 84–94.
- [Clarke 89] CLARKE, LORI A., PODGURSKI, ANDY, RICHARDSSON, DEBRA J., AND ZEIL, STEVEN J. 1989. A Formal Evaluation of Data Flow Path Detection Criteria. *IEEE Trans. on Software Engineering*, Volume SE-15, no. 11 (Nov), pages 1318–1332.
- [Clarke 99] CLARKE, E.M., GRUMBERG, O., AND PELED, D. 1999. *Model Checking*. MIT Press.
- [Clarke 02] CLARKE, D., JÉRON, T., RUSU, V., AND ZINOVIEVA, E. 2002. STG: A Symbolic Test Generation Tool. In *Proc. Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Volume 2280 of *Lecture Notes in Computer Science*. Springer-Verlag, pages 470–475.
- [Cohen 97] COHEN, D.M., DALAL, S.R., FREDMAN, M.L., AND PATTON, G.C. 1997. The AETG system: An approach to testing based on combinatorial design. *IEEE Trans. on Software Engineering*, Volume SE-23, no. 7 (July), pages 437–444.
- [Consel 93] CONSEL, C. AND DANVY, O. 1993. Tutorial Notes on Partial Evaluation. In *Proc. ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM Press, pages 493–501.
- [Cronqvist 04] CRONQVIST, MATS. 2004. Troubleshooting a large ERLANG system. In *Proc. 2004 ACM SIGPLAN workshop on Erlang, ERLANG '04*. ACM, New York, NY, USA, pages 11–15.

- [D'Souza 03] D'SOUZA, DEEPAK AND MUKUND, MADHAVAN. 2003. Checking Consistency of SDL+MSC Specifications. In *Proc. 10th Int. SPIN Workshop*, Volume 2648 of *Lecture Notes in Computer Science*. Springer, pages 151–165.
- [Dijkstra 75] DIJKSTRA, EDSGER W. 1975. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. *Commun. ACM*, Volume 18, no. 8 (Aug.), pages 453–457.
- [TTC 03] ETSI. 2003. *The Testing and Test Control Notation version 3*.
- [El-Far 02] EL-FAR, IBRAHIM K. AND WHITTAKER, J.A. 2002. Model-based software testing. *Encyclopedia of Software Engineering, Volume 1*, pages 825–837.
- [Fernandez 96] FERNANDEZ, J.-C., JARD, C., JÉRON, T., AND VIHO, C. 1996. Using On-the-fly Verification Techniques for the Generation of Test Suites. In *Proc. 8th Int. Conf. on Computer Aided Verification (CAV)*, Volume 1102 of *Lecture Notes in Computer Science*. Springer, pages 348–359.
- [Fernandez 97] FERNANDEZ, J.-C., JARD, C., JÉRON, T., AND VIHO, C. 1997. An Experiment in Automatic Generation of Test Suites for Protocols with Verification Technology. *Science of Computer Programming*, Volume 29, no. 1-2, pages 123–146.
- [Frankl 88] FRANKL, P.G. AND WEYUKER, E. J. 1988. An Applicable Family of Data Flow Testing Criteria. *IEEE Trans. on Software Engineering*, Volume 14, (Oct.), pages 1483–1498.
- [Frantzen 05] FRANTZEN, L., TRETMANS, J., AND WILLEMSE, T. 2005. Test Generation Based on Symbolic Specifications. In *Proc. Int. Workshop on Formal Approaches to Testing of Software (FATES)*, Volume 3395 of *Lecture Notes in Computer Science*. Springer-Verlag, pages 1–15.
- [Fraser 09] FRASER, GORDON, WOTAWA, FRANZ, AND AMMANN, PAUL E. 2009. Testing with Model Checkers: A Survey. *Softw. Test. Verif. Reliab.*, Volume 19, no. 3 (Sept.), pages 215–261.
- [Fredlund 01] FREDLUND, LARS ÅKE. 2001. *A Framework for Reasoning about ERLANG code*. PhD thesis, Royal Institute of Technology, Stockholm.
- [Friedman 02] FRIEDMAN, G., HARTMAN, A., NAGIN, K., AND SHIRAN, T. 2002. Projected State Machine Coverage for Software Testing. In *Proc. Int. Symposium on Software Testing and Analysis (ISSTA)*. ACM, pages 134–143.
- [Gargantini 01] GARGANTINI, A. AND RICCOBENE, E. 2001. ASM-Based Testing: Coverage Criteria and Automatic Test Sequence. *Journal of Universal Computer Science*, Volume 7, no. 11, pages 1050–1067.
- [Gargantini 03] GARGANTINI, A., RICCOBENE, E., AND RINZIVILLO, S. 2003. Using Spin to Generate Tests from ASM Specifications. In *Proc. 10th Int. Workshop on Abstract State Machines (ASM 2003)*, Volume 2589 of *Lecture Notes in Computer Science*. Springer-Verlag, pages 263–277.
- [George 12] GEORGE, DEVARAJ. 2012. *On the effectiveness of specification-based structural test-coverage criteria as test-data*

- generators for safety-critical systems. PhD thesis, University of Minnesota.
- [George 05] GEORGE, D., HEIMDAHL, M., AND LIANG, D. 2005. Coverage-Directed Test Generation with Model Checkers: Challenges and Opportunities. In *COMPSAC 2005, Volume 1*. IEEE Computer Society, pages 455–462.
- [Gnesi 02] GNESI, STEFANIA, LATELLA, DIEGO, AND MASSINK, MIEKE. 2002. Modular semantics for a UML statechart diagrams kernel and its extension to multicharts and branching time model-checking. *Journal of Logic and Algebraic Programming, Volume 51*, no. 1, pages 43–75.
- [Goga 01] GOGA, N. 2001. Comparing TorX, Autolink, TGV and UIO Test Algorithms. In *Proc. 10th International SDL Forum*, Volume 2078 of *Lecture Notes in Computer Science*. Springer-Verlag, pages 379–402.
- [Gotlieb 98] GOTLIEB, ARNAUD, BOTELLA, BERNARD, AND RUEHER, MICHEL. 1998. Automatic test data generation using constraint solving techniques. In *Proc. Int. Symposium on Software Testing and Analysis (ISSTA)*. ACM, pages 53–62.
- [Grabowski 95] GRABOWSKI, J., HOGREFE, D., NUSSBAUMER, I., AND SPICHTIGER, A. 1995. Test Case Specification Based on MSCs and ASN.1. In *SDL'95 with MSC in CASE, Proc. 7th SDL Forum*. Elsevier, Oslo, Norway, pages 307–322.
- [Grieskamp 11] GRIESKAMP, WOLFGANG, KICILLOF, NICOLAS, STOBIE, KEITH, AND BRABERMAN, VICTOR. 2011. Model-based Quality Assurance of Protocol Documentation: Tools and Methodology. *Softw. Test. Verif. Reliab.*, Volume 21, no. 1 (Mar.), pages 55–71.
- [Grindal 07] GRINDAL, MATS. 2007. *Handling Combinatorial Explosion in Software Testing*. PhD thesis, Linköping University, Linköping, Sweden.
- [Havelund 02] HAVELUND, K. AND ROSU, G. 2002. Synthesizing Monitors for Safety Properties. In *Proc. Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Volume 2280 of *Lecture Notes in Computer Science*. Springer-Verlag, pages 324–356.
- [Heimdahl 04] HEIMDAHL, MATS P.E., GEORGE, DEVARAJ, AND WEBER, ROBERT. 2004. Specification Test Coverage Adequacy Criteria = Specification Test Generation Inadequacy Criteria? In *8th IEEE Int. Symp. on High-Assurance Systems Engineering (HASE)*. IEEE Computer Society, pages 178–186.
- [Herman 76] HERMAN, P.M. 1976. A data flow analysis approach to program testing. *Australian Computer J.*, Volume 8, no. 3 (Nov.), pages 92–96.
- [Hessel 07] HESSEL, A. 2007. *Model-Based Test Case Generation for Real-Time Systems*. PhD thesis, Dept. of Information Technology, Uppsala University, Sweden, Uppsala, Sweden.
- [Hessel 08] HESSEL, ANDERS, LARSEN, KIM GULDSTRAND, MIKUCIONIS, MARIUS, NIELSEN, BRIAN, PETTERSSON, PAUL, AND

- SKOU, ARNE. 2008. Testing Real-Time Systems Using UPPAAL. In *Formal Methods and Testing, An Outcome of the FORTEST Network, Revised Selected Papers*, Volume 4949 of *Lecture Notes in Computer Science*. Springer-Verlag, pages 77–117.
- [Holzmann 97] HOLZMANN, G.J. 1997. The Model Checker SPIN. *IEEE Trans. on Software Engineering*, Volume SE-23, no. 5 (May), pages 279–295.
- [Holzmann 03] HOLZMANN, G.J. 2003. *The SPIN Model Checker*. Addison-Wesley.
- [Holzmann 02] HOLZMANN, G.J. AND SMITH, H. 2002. An Automated Verification Method for Distributed Systems Software Based on Model Extraction. *IEEE Trans. on Software Engineering*, Volume 28, pages 1–14.
- [Hong 02] HONG, H.S., LEE, I., SOKOLSKY, O., AND URAL, H. 2002. A Temporal Logic Based Theory of Test Coverage and Generation. In *Proc. Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Volume 2280 of *Lecture Notes in Computer Science*. Springer-Verlag, pages 327–341.
- [Hong 03] HONG, H.S., CHA, S.D., LEE, I., SOKOLSKY, O., AND URAL, H. 2003. Data Flow Testing as Model Checking. In *Proc. Int. Conf. on Software Engineering (ICSE)*. IEEE Computer Society, pages 232–242.
- [Hughes 10] HUGHES, JOHN. 2010. Software testing with QuickCheck. In *Central European Functional Programming School*. Springer-Verlag, pages 183–223.
- [Huima 07] HUIMA, ANTTI. 2007. Implementing Conformiq Qtronic. In *Proc. TestCom/FATES*, Volume 4581 of *Lecture Notes in Computer Science*. Springer-Verlag, pages 1–12.
- [IBM 11] IBM. 2011. Rational Tau.
- [ITU-T 99a] ITU-T. 1999. Recommendation Z.120, Message Sequence Charts (MSC).
- [ITU-T 99b] ITU-T. 1999. Recommendation Z.100, Specification and Description Language (SDL).
- [Ip 99] IP, C. NORRIS AND DILL, DAVID L. 1999. Verifying Systems with Replicated Components in $\text{Mur}\varphi$. *Formal Methods in System Design*, Volume 14, no. 3 (May), pages 147–158.
- [Jacky 08] JACKY, JONATHAN, VEANES, MARGUS, CAMPBELL, COLIN, AND SCHULTE, WOLFRAM. 2008. *Model-Based Software Testing and Analysis with C#*. Cambridge University Press.
- [Janhunen 11] JANHUNEN, TOMI, NIEMELÄ, ILKKA, OETSCH, JOHANNES, PÜHRER, JÖRG, AND TOMPITS, HANS. 2011. Random vs. Structure-Based Testing of Answer-Set Programs: An Experimental Comparison. In *Logic Programming and Nonmonotonic Reasoning*. Volume 6645 of *Lecture Notes in Computer Science*. Springer-Verlag, pages 242–247.
- [Jantsch 00] JANTSCH, AXEL AND SANDER, INGO. 2000. On the Roles of Functions and Objects in System Specification. In *Proc. Int.*

- Workshop on Hardware/Software Codesign (CODES)*. ACM, pages 8–12.
- [Jantsch 98] JANTSCH, A., KUMAR, S., SANDER, I., SVANTESSON, B., OBERG, J., HEMANI, A., ELLERVEE, P., AND O’NILS, M. 1998. Comparison of six languages for system level descriptions of telecom systems. In *First Int. Forum on Design Languages*, Volume 2. Lausanne, Switzerland, pages 139–148.
- [Jard 05] JARD, C. AND JERON, T. 2005. TGV: theory, principles and algorithms. *Int. Journal on Software Tools for Technology Transfer*, Volume 7, no. 4, pages 297–315.
- [Kadono 09] KADONO, MASAYA, TSUCHIYA, TATSUHIRO, AND KIKUNO, TOHRU. 2009. Using the NuSMV Model Checker for Test Generation from Statecharts. In *Proc. 2009 15th IEEE Pacific Rim Int. Symposium on Dependable Computing*, PRDC ’09. IEEE Computer Society, Washington, DC, USA, pages 37–42.
- [King 76] KING, JAMES C. 1976. Symbolic Execution and Program Testing. *Communications of the ACM*, Volume 19, no. 7 (July), pages 385–394.
- [Knoop 96] KNOOP, J., STEFFEN, B., AND VOLLMER, J. 1996. Parallelism for Free: Efficient and Optimal Bitvector Analyses for Parallel Programs. *ACM Transactions on Programming Languages and Systems*, Volume 18, no. 3, pages 268–299.
- [Koch 98] KOCH, BEAT, GRABOWSKI, JENS, HOGREFE, DIETER, AND SCHMITT, MICHAEL. 1998. Autolink - A Tool for Automatic Test Generation from SDL Specifications. In *Proc. Workshop on Industrial Strength Formal Specication Techniques (WIFT’98)*. IEEE Computer Society, pages 21–23.
- [Lai 02] LAI, R. 2002. A survey of communication protocol testing. *Journal of Systems and Software*, Volume 62, pages 21–46.
- [Larsen 97] LARSEN, K.G., PETTERSSON, P., AND YI, W. 1997. UPPAAL in a Nutshell. *Software Tools for Technology Transfer*, Volume 1, no. 1-2, pages 134–152.
- [Laski 83] LASKI, J. W. AND KOREL, B. 1983. A Data Flow Oriented Program Testing Strategy. *IEEE Trans. on Software Engineering*, Volume SE-9, no. 3 (May), pages 347–354.
- [Leavens 99] LEAVENS, GARY T., BAKER, ALBERT L., AND RUBY, CLYDE. 1999. Preliminary design of JML: A behavioral interface specification language for Java. Technical report, Techn. Rep. 98-06c, Dep. of Comp. Sci., Iowa State Univ. (<http://www.cs.iastate.edu>).
- [Lin 96] LIN, HUIMIN. 1996. Symbolic transition graph with assignment. In *Proc. Int. Conf. on Concurrency Theory (CONCUR)*, Volume 1119 of *Lecture Notes in Computer Science*. Springer-Verlag, pages 50–65.
- [Lindahl 06] LINDAHL, TOBIAS AND SAGONAS, KONSTANTINOS. 2006. Practical type inference based on success typings. In *Proc. 8th ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP)*. ACM Press, New York, NY, USA, pages 167–178.

- [Malaiya 02] MALAIYA, YASHWANT K., LI, MICHAEL NAIXIN, BIEMAN, JAMES M., AND KARCICH, RICK. 2002. Software Reliability Growth with Test Coverage. *IEEE Transactions on Reliability*, Volume 51, pages 420–426.
- [Mandl 85] MANDL, O. 1985. Orthogonal Latin Squares: An experiment design to compiler testing. *Communications of the ACM*, Volume 28, no. 10 (Oct.), pages 1054–1058.
- [Mandrioli 95] MANDRIOLI, D., MORASCA, S., AND MORZENTI, A. 1995. Generating Test Cases for Real-Time Systems from Logic Specifications. *ACM Trans. on Computer Systems*, Volume 13, no. 4 (Nov.), pages 365–398.
- [Marinescu 15] MARINESCU, RALUCA, SECELEANU, CRISTINA, GUEN, H L NE LE, AND PETTERSSON, PAUL. 2015. A Research Overview of Tool-Supported Model-based Testing of Requirements-based Designs. *Advances in Computers*, Volume 98, pages 89 – 140.
- [Mauw 97] MAUW, S. AND RENIERS, M.A. 1997. Operational Semantics for MSC'96. In *SDL'97 - Time for Testing - SDL, MSC and Trends*. Elsevier, pages 135–152.
- [Myers 79] MYERS, GLENFORD J. 1979. *Art of Software Testing*. John Wiley & Sons, Inc., New York, NY, USA.
- [Mäkinen 07] MÄKINEN, M. A. 2007. Model Based Approach to Software Testing. Master's thesis, Helsinki University of Technology.
- [Ntafos 88] NTAPOS, S. C. 1988. A Comparison of Some Structural Testing Strategies. *IEEE Trans. Softw. Eng.*, Volume 14, no. 6, pages 868–874.
- [Ntafos 01] NTAPOS, SIMEON C. 2001. On Comparisons of Random, Partition, and Proportional Partition Testing. *IEEE Trans. Softw. Eng.*, Volume 27, no. 10, pages 949–960.
- [Plotkin 81] PLOTKIN, G. 1981. A Structural Approach to Operational Semantics. Technical report DAIMI FN-19, Computer Science Department, Aarhus University, Denmark.
- [Prenninger 05] PRENNINGER, W. AND PRETSCHNER, A. 2005. Abstractions for Model-Based Testing. *Electronic Notes in Theoretical Computer Science*, Volume 116, (Jan), pages 59–71.
- [Pretschner 01] PRETSCHNER, A. 2001. Classical search strategies for test case generation with constraint logic programming. In *Proc. Formal Approaches to Testing of Software, FATES '01*. Aalborg, Denmark, pages 47–60.
- [Pretschner 05] PRETSCHNER, A., PRENNINGER, W., WAGNER, S., KÜHNEL, C., BAUMGARTNER, M., SOSTAWA, B., ZÖLCH, R., AND STAUNER, T. 2005. One evaluation of model-based testing and its automation. In *Proc. Int. Conf. on Software Engineering (ICSE)*. ACM, New York, NY, USA, pages 392–401.
- [Rapin 03] RAPIN, N., GASTON, C., LAPITRE, A., AND GALLOIS, J.-P. 2003. Behavioural Unfolding of Formal Specifications Based on Communicating extended automata. In *Proc. 1st Int. Workshop on*

- Automated Technology for Verification and Analysis*. Tapei, Taiwan.
- [Rapps 85] RAPPS, SANDRA AND WEYUKER, ELAINE J. 1985. Selecting software test data using data flow information. *IEEE Trans. on Software Engineering*, Volume 11, no. 4 (Apr.), pages 367–375.
- [Rayadurgam 01] RAYADURGAM, S. AND HEIMDAHL, M. P. 2001. Test-Sequence Generation from Formal Requirements Models. In *Proc. 6th IEEE Int. Symposium on High-Assurance Systems Engineering (HASE 2001)*. IEEE Computer Society Press, pages 23–31.
- [Richardson 92] RICHARDSON, D.J., LEIF-AHA, S., AND O’MALLEY, T.O. 1992. Specification-Based Test Oracles for Reactive Systems. In *Proc. Int. Conf. on Software Engineering (ICSE)*. ACM Press, pages 105–118.
- [Rusu 00] RUSU, V., DU BOUSQUET, L., AND JÉRON, T. 2000. An Approach to Symbolic Test Generation. In *Proc. Int. Conf. Integrated Formal Methods (IFM)*, Volume 1945 of *Lecture Notes in Computer Science*. Springer-Verlag, pages 338–357.
- [Santos-Neto 08] SANTOS-NETO, PEDRO, RESENDE, RODOLFO F., AND PÁDUA, CLARINDO. 2008. An evaluation of a model-based testing method for information systems. In *Proc. 2008 ACM symposium on Applied computing*, SAC ’08. ACM, New York, NY, USA, pages 770–776.
- [Sarma 10] SARMA, MONALISA, MURTHY, P. V. R., JELL, SYLVIA, AND ULRICH, ANDREAS. 2010. Model-based testing in industry: a case study with two MBT tools. In *Proc. 5th Workshop on Automation of Software Test*, AST ’10. ACM, New York, NY, USA, pages 87–90.
- [Shafique 15] SHAFIQUE, MUHAMMAD AND LABICHE, YVAN. 2015. A systematic review of state-based test tools. *Int. Journal on Software Tools for Technology Transfer*, Volume 17, no. 1, pages 59–76.
- [Sinha 06] SINHA, A., WILLIAMS, C. E., AND SANTHANAM, P. 2006. A measurement framework for evaluating model-based test generation tools. *IBM Systems Journal*, Volume 45, no. 3, pages 501–514.
- [Sommerville 10] SOMMERVILLE, IAN. 2010. *Software Engineering*, 9 edition. Addison Wesley.
- [Staats 10] STAATS, MATT AND PĂȘĂREANU, CORINA. 2010. Parallel symbolic execution for structural test generation. In *Proc. Int. Symposium on Software Testing and Analysis (ISSTA)*. ACM, New York, NY, USA, pages 183–194.
- [Staats 12] STAATS, M., GAY, G., WHALEN, M., AND HEIMDAHL, M. 2012. On the danger of coverage directed test case generation. In *Fundamental Approaches to Software Engineering*. Volume 7212 of *Lecture Notes in Computer Science*. Springer-Verlag, pages 409–424.
- [Sving 10] SVING, ROBIN AND ÖMAN, PETER. 2010. Pilot Project for Model Based Testing using Conformiq Qtronic. Master’s thesis, University of Uppsala.
- [Thorup 10] THORUP, K. K. 2010. Triq.
- [Tretmans 96] TRETMANS, J. 1996. Test Generation with Inputs, Outputs,

- and Quiescence. *Software – Concepts and Tools*, Volume 17, no. 3, pages 103–120.
- [Tretmans 02] TRETMAINS, J. AND BRINKSMA, E. 2002. Côte de Resysste – Automated Model Based Testing. In *Progress 2002 – 3rd Workshop on Embedded Systems*. STW Technology Foundation, Utrecht, The Netherlands, pages 246–255.
- [Tretmans 03] TRETMAINS, JAN AND BRINKSMA, ED. 2003. TorX: Automated Model-Based Testing. In *First European Conference on Model-Driven Software Engineering*, pages 31–43.
- [Utting 08] UTTING, M. 2008. The Role of Model-Based Testing. In *Conf. Verified Software: Theories, Tools, Experiments (VSTTE) 2005, Revised Selected Papers and Discussions*, Volume 4171 of *Lecture Notes in Computer Science*. Springer-Verlag, Zürich, Switzerland, pages 510–517.
- [Utting 07] UTTING, MARK AND LEGEARD, BRUNO. 2007. *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.
- [Utting 12] UTTING, MARK, PRETSCHNER, ALEXANDER, AND LEGEARD, BRUNO. 2012. A taxonomy of model-based testing approaches. *Software Testing, Verification and Reliability*, Volume 22, no. 5 (Aug.), pages 297–312.
- [Vardi 86] VARDI, M. Y. AND WOLPER, P. 1986. An automata-theoretic approach to automatic program verification. In *Proc. Symp. on Logic in Computer Science (LICS)*. IEEE Computer Society, pages 332–344.
- [Veanes 08] VEANES, MARGUS, CAMPBELL, COLIN, GRIESKAMP, WOLFGANG, SCHULTE, WOLFRAM, TILLMANN, NIKOLAI, AND NACHMANSON, LEV. 2008. Model-Based Testing of Object-Oriented Reactive Systems with Spec Explorer. In *Formal Methods and Testing, An Outcome of the FORTEST Network, Revised Selected Papers*. Volume 4949 of *Lecture Notes in Computer Science*. Springer-Verlag, pages 39–76.
- [Vilkomir 01] VILKOMIR, S. A. AND BOWEN, J. P. 2001. Formalization of software testing criteria using the Z notation. In *Proc. 25th Int. Computer Software and Applications Conference (COMPSAC 2001)*. IEEE Computer Society Press, pages 351–356.
- [Vos 12] VOS, TANJA E.J., BAARS, ARTHUR I., LINDLAR, FELIX F., WINDISCH, ANDREAS, WILMES, BENJAMIN, GROSS, HAMILTON, KRUSE, PETER M., AND WEGENER, JOACHIM. 2012. Industrial Case Studies for Evaluating Search Based Structural Testing. *Int. Journal of Software Engineering and Knowledge Engineering*, Volume 22, no. 08, pages 1123–1149.
- [Vries 00] VRIES, RENÉ G. DE AND TRETMAINS, JAN. 2000. On-the-fly Conformance Testing using SPIN. *Int. Journal on Software Tools for Technology Transfer*, Volume 2, no. 4, pages 382–393.
- [Weiglhofer 09] WEIGLHOFER, MARTIN, AICHERNIG, BERNHARD K., AND WOTAWA, FRANZ. 2009. Fault-Based Conformance Testing in

- Practice. *Int. J. Software and Informatics*, Volume 3, no. 2-3, pages 375–411.
- [Weißleder 10] WEISSLEDER, STEPHAN. 2010. *Test models and coverage criteria for automatic model-based test generation with UML state machines..* PhD thesis, Humboldt University of Berlin.
- [Widera 04] WIDERA, M. 2004. Flow Graphs for Testing Sequential ERLANG Programs. In *Proc. 2004 ACM SIGPLAN workshop on Erlang*. Snowbird, Utah, USA.
- [Wille 82] WILLE, RUDOLF. 1982. Restructuring lattice theory: an approach based on hierarchies of concepts. In *Ordered sets*. Reidel, Dordrecht–Boston, pages 445–470.
- [3gp 99] 3GPP. 1999. *Mobile Application Part (MAP) specification*.
- [3gp 00] 3GPP. 2000. *Functional stage 2 description of Location Services (LCS)*.
- [Erl 15] ERICSSON. 2015. *Erlang Open Telecom Platform*.
- [oma 04] OMA. 2004. *Mobile Location Protocol*.
- [OMG 03] OMG. 2003. *Unified Modeling Language, UML*.

Acta Universitatis Upsaliensis

Uppsala Dissertations from the Faculty of Science

Editor: The Dean of the Faculty of Science

1–11: 1970–1975

12. *Lars Thofelt*: Studies on leaf temperature recorded by direct measurement and by thermography. 1975.
13. *Monica Henricsson*: Nutritional studies on *Chara globularis* Thuill., *Chara zeylanica* Willd., and *Chara haitensis* Turpin. 1976.
14. *Göran Kloow*: Studies on Regenerated Cellulose by the Fluorescence Depolarization Technique. 1976.
15. *Carl-Magnus Backman*: A High Pressure Study of the Photolytic Decomposition of Azoethane and Propionyl Peroxide. 1976.
16. *Lennart Källströmer*: The significance of biotin and certain monosaccharides for the growth of *Aspergillus niger* on rhamnose medium at elevated temperature. 1977.
17. *Staffan Renlund*: Identification of Oxytocin and Vasopressin in the Bovine Adenohypophysis. 1978.
18. *Bengt Finnström*: Effects of pH, Ionic Strength and Light Intensity on the Flash Photolysis of L-tryptophan. 1978.
19. *Thomas C. Amu*: Diffusion in Dilute Solutions: An Experimental Study with Special Reference to the Effect of Size and Shape of Solute and Solvent Molecules. 1978.
20. *Lars Tegnér*: A Flash Photolysis Study of the Thermal Cis-Trans Isomerization of Some Aromatic Schiff Bases in Solution. 1979.
21. *Stig Tormod*: A High-Speed Stopped Flow Laser Light Scattering Apparatus and its Application in a Study of Conformational Changes in Bovine Serum Albumin. 1985.
22. *Björn Varnestig*: Coulomb Excitation of Rotational Nuclei. 1987.
23. *Frans Lettenström*: A study of nuclear effects in deep inelastic muon scattering. 1988.
24. *Göran Ericsson*: Production of Heavy Hypernuclei in Antiproton Annihilation. Study of their decay in the fission channel. 1988.
25. *Fang Peng*: The Geopotential: Modelling Techniques and Physical Implications with Case Studies in the South and East China Sea and Fennoscandia. 1989.
26. *Md. Anowar Hossain*: Seismic Refraction Studies in the Baltic Shield along the Fennolora Profile. 1989.
27. *Lars Erik Svensson*: Coulomb Excitation of Vibrational Nuclei. 1989.
28. *Bengt Carlsson*: Digital differentiating filters and model based fault detection. 1989.
29. *Alexander Edgar Kavka*: Coulomb Excitation. Analytical Methods and Experimental Results on even Selenium Nuclei. 1989.
30. *Christopher Juhlin*: Seismic Attenuation, Shear Wave Anisotropy and Some Aspects of Fracturing in the Crystalline Rock of the Siljan Ring Area, Central Sweden. 1990.

31. *Torbjörn Wigren*: Recursive Identification Based on the Nonlinear Wiener Model. 1990.
32. *Kjell Janson*: Experimental investigations of the proton and deuteron structure functions. 1991.
33. *Suzanne W. Harris*: Positive Muons in Crystalline and Amorphous Solids. 1991.
34. *Jan Blomgren*: Experimental Studies of Giant Resonances in Medium-Weight Spherical Nuclei. 1991.
35. *Jonas Lindgren*: Waveform Inversion of Seismic Reflection Data through Local Optimisation Methods. 1992.
36. *Liqi Fang*: Dynamic Light Scattering from Polymer Gels and Semidilute Solutions. 1992.
37. *Raymond Munier*: Segmentation, Fragmentation and Jostling of the Baltic Shield with Time. 1993.

Prior to January 1994, the series was called *Uppsala Dissertations from the Faculty of Science*.

Acta Universitatis Upsaliensis

Uppsala Dissertations from the Faculty of Science and Technology

Editor: The Dean of the Faculty of Science

- 1–14: 1994–1997. 15–21: 1998–1999. 22–35: 2000–2001. 36–51: 2002–2003.
52. *Erik Larsson*: Identification of Stochastic Continuous-time Systems. Algorithms, Irregular Sampling and Cramér-Rao Bounds. 2004.
53. *Per Åhgren*: On System Identification and Acoustic Echo Cancellation. 2004.
54. *Felix Wehrmann*: On Modelling Nonlinear Variation in Discrete Appearances of Objects. 2004.
55. *Peter S. Hammerstein*: Stochastic Resonance and Noise-Assisted Signal Transfer. On Coupling-Effects of Stochastic Resonators and Spectral Optimization of Fluctuations in Random Network Switches. 2004.
56. *Esteban Damián Avendaño Soto*: Electrochromism in Nickel-based Oxides. Coloration Mechanisms and Optimization of Sputter-deposited Thin Films. 2004.
57. *Jenny Öhman Persson*: The Obvious & The Essential. Interpreting Software Development & Organizational Change. 2004.
58. *Chariklia Rouki*: Experimental Studies of the Synthesis and the Survival Probability of Transactinides. 2004.
59. *Emad Abd-Elrady*: Nonlinear Approaches to Periodic Signal Modeling. 2005.
60. *Marcus Nilsson*: Regular Model Checking. 2005.
61. *Pritha Mahata*: Model Checking Parameterized Timed Systems. 2005.
62. *Anders Berglund*: Learning computer systems in a distributed project course: The what, why, how and where. 2005.
63. *Barbara Piechocinska*: Physics from Wholeness. Dynamical Totality as a Conceptual Foundation for Physical Theories. 2005.
64. *Pär Samuelsson*: Control of Nitrogen Removal in Activated Sludge Processes. 2005.

65. *Mats Ekman*: Modeling and Control of Bilinear Systems. Application to the Activated Sludge Process. 2005.
66. *Milena Ivanova*: Scalable Scientific Stream Query Processing. 2005.
67. *Zoran Radovic*: Software Techniques for Distributed Shared Memory. 2005.
68. *Richard Abrahamsson*: Estimation Problems in Array Signal Processing, System Identification, and Radar Imagery. 2006.
69. *Fredrik Robelius*: Giant Oil Fields – The Highway to Oil. Giant Oil Fields and their Importance for Future Oil Production. 2007.
70. *Anna Davour*: Search for low mass WIMPs with the AMANDA neutrino telescope. 2007.
71. *Magnus Ågren*: Set Constraints for Local Search. 2007.
72. *Ahmed Rezine*: Parameterized Systems: Generalizing and Simplifying Automatic Verification. 2008.
73. *Linda Brus*: Nonlinear Identification and Control with Solar Energy Applications. 2008.
74. *Peter Nauchler*: Estimation and Control of Resonant Systems with Stochastic Disturbances. 2008.
75. *Johan Petrini*: Querying RDF Schema Views of Relational Databases. 2008.
76. *Noomene Ben Henda*: Infinite-state Stochastic and Parameterized Systems. 2008.
77. *Samson Keleta*: Double Pion Production in $dd \rightarrow \alpha\pi\pi$ Reaction. 2008.
78. *Mei Hong*: Analysis of Some Methods for Identifying Dynamic Errors-invariables Systems. 2008.
79. *Robin Strand*: Distance Functions and Image Processing on Point-Lattices With Focus on the 3D Face-and Body-centered Cubic Grids. 2008.
80. *Ruslan Fomkin*: Optimization and Execution of Complex Scientific Queries. 2009.
81. *John Airey*: Science, Language and Literacy. Case Studies of Learning in Swedish University Physics. 2009.
82. *Arvid Pohl*: Search for Subrelativistic Particles with the AMANDA Neutrino Telescope. 2009.
83. *Anna Danielsson*: Doing Physics – Doing Gender. An Exploration of Physics Students' Identity Constitution in the Context of Laboratory Work. 2009.
84. *Karin Schöning*: Meson Production in pd Collisions. 2009.
85. *Henrik Petré*: η Meson Production in Proton-Proton Collisions at Excess Energies of 40 and 72 MeV. 2009.
86. *Jan Henry Nyström*: Analysing Fault Tolerance for ERLANG Applications. 2009.
87. *John Håkansson*: Design and Verification of Component Based Real-Time Systems. 2009.
88. *Sophie Grape*: Studies of PWO Crystals and Simulations of the $\bar{p}p \rightarrow \bar{\Lambda}\Lambda, \bar{\Lambda}\Sigma^0$ Reactions for the PANDA Experiment. 2009.
90. *Agnes Rensfelt*: Viscoelastic Materials. Identification and Experiment Design. 2010.
91. *Erik Gudmundson*: Signal Processing for Spectroscopic Applications. 2010.
92. *Björn Halvarsson*: Interaction Analysis in Multivariable Control Systems. Applications to Bioreactors for Nitrogen Removal. 2010.
93. *Jesper Bengtson*: Formalising process calculi. 2010.
94. *Magnus Johansson*: Psi-calculi: a Framework for Mobile Process Calculi. Cook your own correct process calculus – just add data and logic. 2010.
95. *Karin Rathsman*: Modeling of Electron Cooling. Theory, Data and Applications. 2010.

96. *Liselott Dominicus van den Bussche*. Getting the Picture of University Physics. 2010.
97. *Olle Engdegård*. A Search for Dark Matter in the Sun with AMANDA and IceCube. 2011.
98. *Matthias Hudl*. Magnetic materials with tunable thermal, electrical, and dynamic properties. An experimental study of magnetocaloric, multiferroic, and spin-glass materials. 2012.
99. *Marcio Costa*. First-principles Studies of Local Structure Effects in Magnetic Materials. 2012.
100. *Patrik Adlarson*. Studies of the Decay $\eta \rightarrow \pi^+ \pi^- \pi^0$ with WASA-at-COSY. 2012.
101. *Erik Thomé*. Multi-Strange and Charmed Antihyperon-Hyperon Physics for PANDA. 2012.
102. *Anette Löfström*. Implementing a Vision. Studying Leaders' Strategic Use of an Intranet while Exploring Ethnography within HCI. 2014.