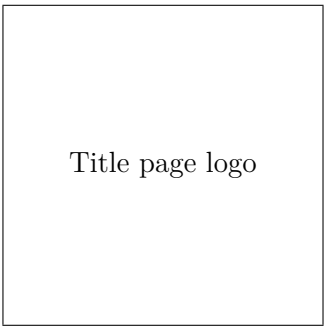
Uppsala Dissertations from the Faculty of Science and Technology

Johan Blom

Model-Based Protocol Testing in an Erlang Environment



To Anna and Emilia

Contents

List of Tables

Acknowledgments

TBD.

Thank you for all the help!

Publications by the Author

Automated Test Generation for Industrial Erlang Applications

[Blom 03] describes an early version of the test suite generation technique and case study, further extended on in this thesis.
Main author.

Specifying and Generating Test Cases Using Observer Automata

[Blom 04] presents a technique for specifying coverage criteria and an algorithm for generating test suites. The presented observer automata is further extended in this thesis to observe symbolic execution.
Co-author, contributed in discussions and writing of the whole paper.

Industrial Evaluation of Test Suite Generation Strategies for Model-Based Testing

[Blom 16] describes the case study, further extended on in this thesis.
Main author.

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

Using Temporal Logic for Modular Specification of Telephone Services

[Blom 94] is an approach for using linear time temporal logic to specify features, such that feature interaction can be detected.
Co-author, contributed in discussions and writing of parts of the paper.

Automatic Detection of Feature Interactions in Temporal Logic

[Blom 95] extends the framework presented in [Blom 94]. This paper presents a formal definition of feature interaction in a linear time temporal logic framework.
Co-author, contributed in discussions and writing of the whole paper.

Creation of Dependent Features

[Blom 96a] is a comprehensive summary, covering our work on feature interaction done to date.
Co-author, contributed in discussions and writing of the whole paper.

Constraint Oriented Temporal Logic Specification

[Blom 96b] is a contribution a larger comparison between specification formalisms and techniques. The specification object was an RPC memory structured as three interacting components. Our approach was based on temporal logic and also included graphical representation of each component of the RPC memory.
Co-author, contributed in discussions and writing of the whole paper.

Formalization of Requirements with Emphasis on Feature Interaction Detection

[Blom 97] use the temporal logic framework in [Blom 95]. This paper presents an approach to organizing specifications into components that could be easily reused and replaced.

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, Broy 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 refered 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

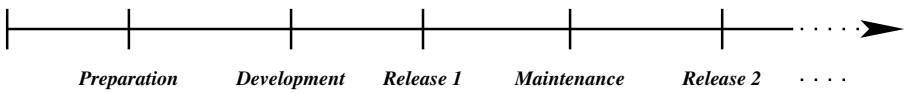


Figure 1.1. Simplified description of the life-cycle of a larger software system.

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 succeded 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 of 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 form 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 respones and configuriation 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 $a(d_1, \ldots, d_k)$, where a is an (input or output) *event type*, and $d_1, \ldots, d_k$ 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.

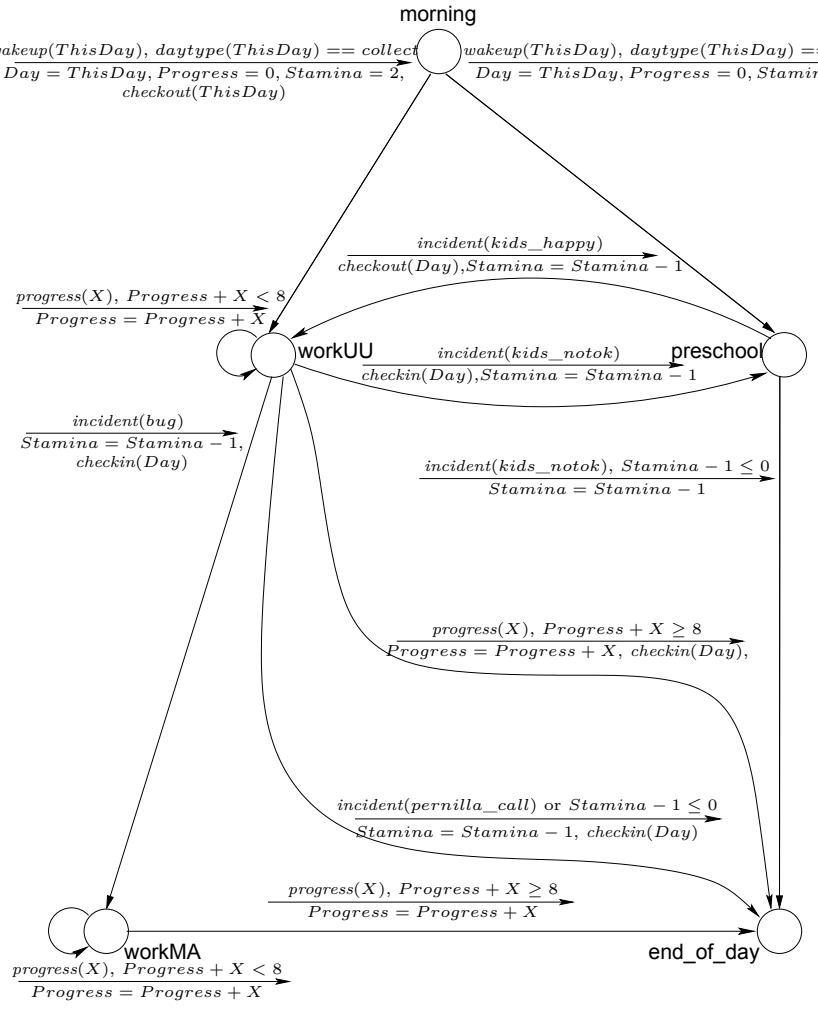


Figure 1.2. A graphical representation of the state machine described in Example 1.1. The complete ERLANG/EFSM specification is given in Figure 2.2.

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 is X is an integer in the range $1, \ldots, 3$, 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 visting 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 immediately 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 $progess(X)$ input event expression, for some X in the range $1, \ldots, 3$.

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.

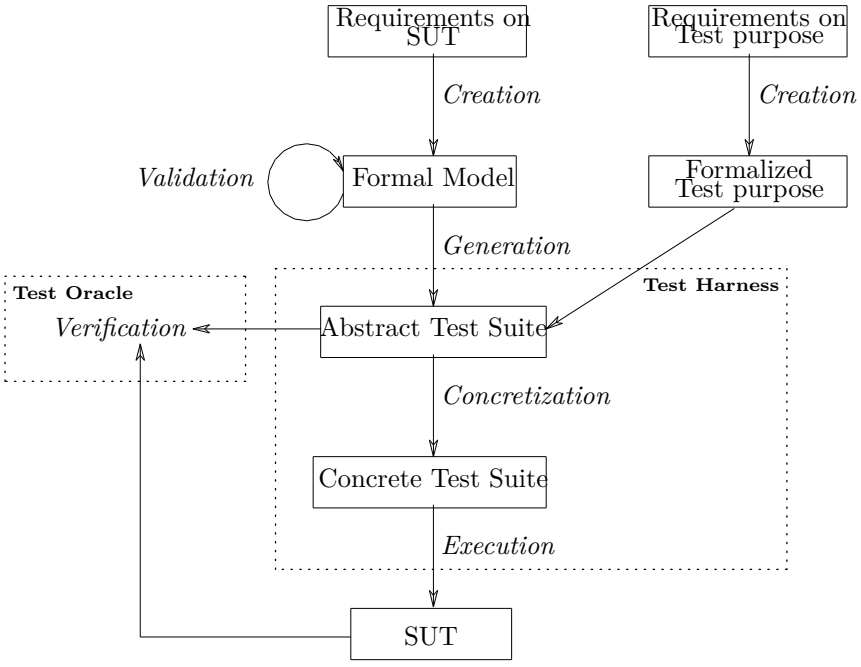


Figure 1.3. An overview of typical components in model-based testing.

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 05], 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.

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 cup of coffee costs 1 unit, so whenever at least 1 unit is inserted and the

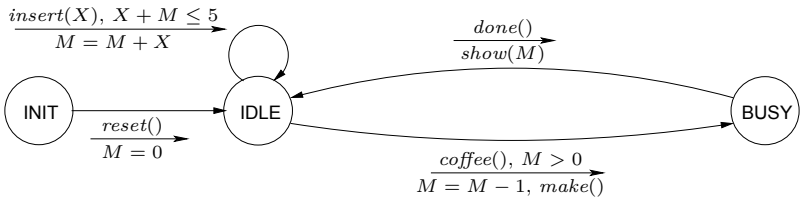


Figure 1.4. An abstract model specifying the controller of a simple coffee machine.

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.

□

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

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

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

$$\left\langle \begin{array}{lllllllll} \mathit{reset}()/ & \rightarrow & \mathit{insert}(p_1)/ & \rightarrow & \mathit{coffee}()/\mathit{make}() & \rightarrow & \mathit{done}()/\mathit{show}() \\ & \rightarrow & \mathit{insert}(p_2)/ & \rightarrow & \mathit{coffee}()/\mathit{make}() & \rightarrow & \mathit{done}()/\mathit{show}() \end{array} , \begin{pmatrix} & p_1 + 0 \leq 5 \\ \wedge & p_1 > 0 \\ \wedge & p_2 + 0 \leq 5 \\ \wedge & p_2 > 0 \end{pmatrix} \right\rangle$$

A challenge with test suite generation is to generate a test suite that increase confidence in the correcteness 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 subdomains and each test case is selected to ensure coverage of some subdomain. 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, $p1$,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 $3*3+2*3+2*3 = 21$ 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 latetely been adopted to model-based testing [Aichernig 12] where test cases covering a mutated formal specification specification are generated. Mutation based testing is based on the hyptheses that: (1) test cases that detect simple faults are likely to also detect complex faults and (2) SUT's are close to beeing 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. Occurence of a *insert*(X) event in a test case could then cause a test environmnent to trigger a (real) insertion of X unitis 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, de Vries 98]. 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, Broy 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.

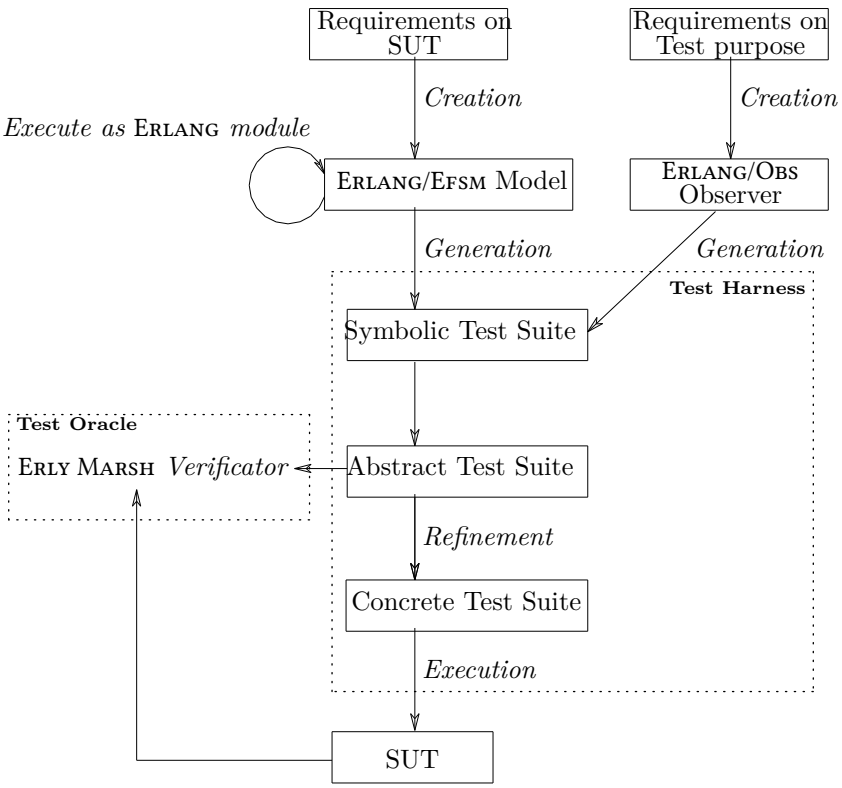


Figure 1.5. Components in model-based testing as used in this thesis.

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 formel model written in ERLANG/EFSM can be accompished 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/EFSM. We have also incorporated the observer mechanism for defining coverage criteria and investigated different strategies for instantiating symbolic parameters.

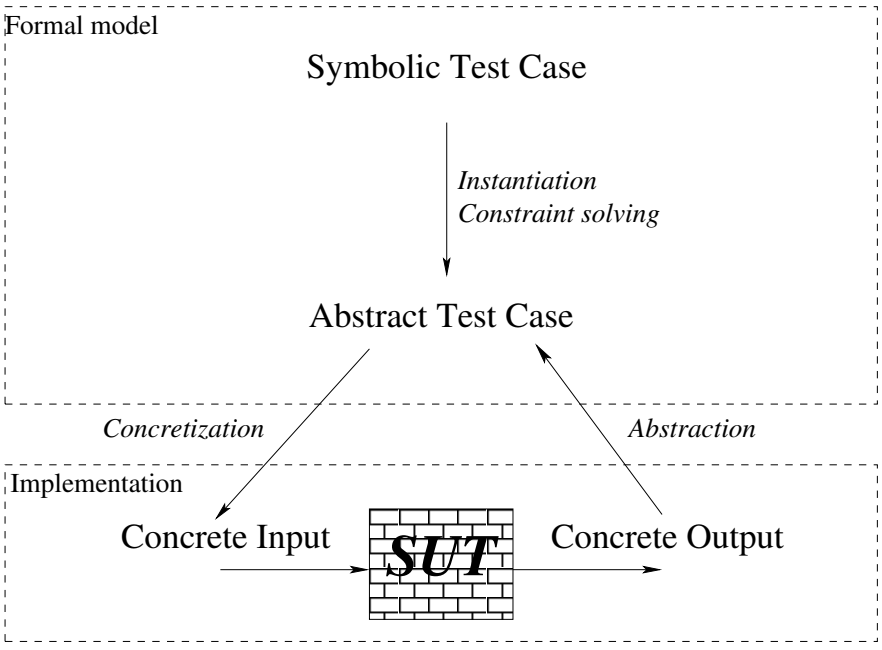


Figure 1.6. An outline of the abstraction levels used in this thesis.

1.6.4 Concretization of Generated Test Cases

Since an ERLANG/EFSM 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 based on the functional language ERLANG, primarily intended for automated test suite generation. We will refer to this language as ERLANG/EFSM.

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 a 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 replace 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 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 [Eri 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 tranforms an Erlang/Efsm specification into the form of a `gen_fsm` behavior, which is directly executable.

The rest of this section is organized as follows. Section 2.2 introduces Erlang/Efsm by an example. Section 2.3 describes the syntax and intended meaning of Erlang/Efsm, and Section 2.4 describes its operational semantics, Section 2.5 explain how Erlang/Efsm specifications define state machines, Section 2.7 describes the corresponding operational semantics for symbolic execution, and Section 2.9 define the semantics for a symbolic state machine. The transformation of Erlang/Efsm specifications into executable form is defined in Section 2.11 where we further elaborate on the relation between the Erlang/Efsm model and the Erlang/Otp `gen_fsm` behavior. All Erlang preprocessor extensions added in Erlang/Efsm are further explained in Section 2.3.2.

2.2 An Overview of Erlang/Efsm

In this section, we introduce the overall structure of Erlang/Efsm specficiations, 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 exressions, output expressions and user defined functions are optional, and
- for each control location, a list of *transition clauses* represent all edges in the state machine.

Before we explain Erlang/Efsm language in more detail, we illustrate with 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],
```

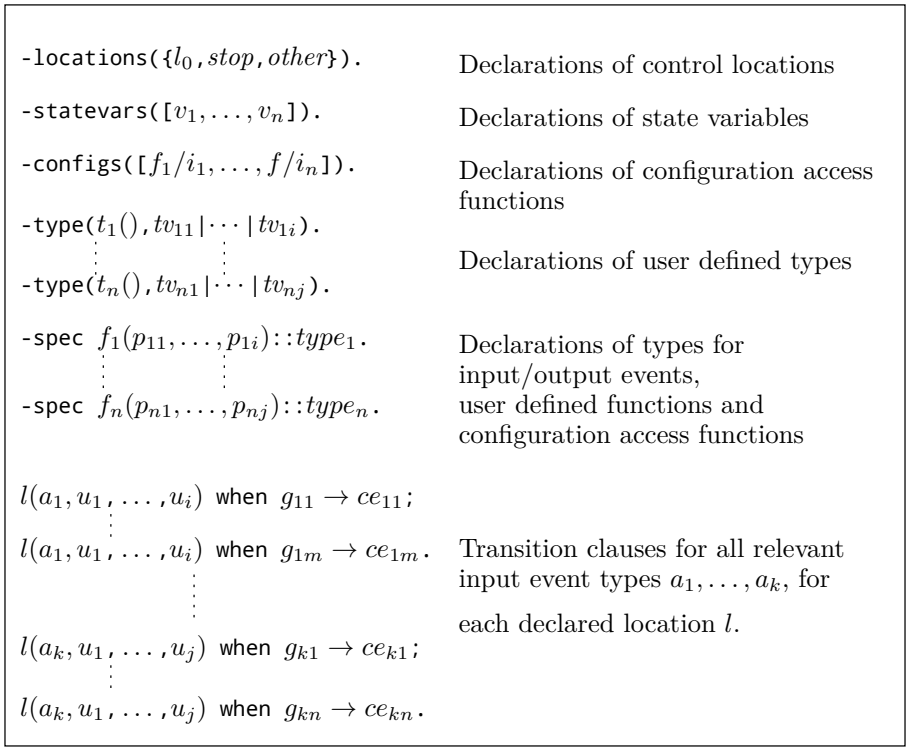


Figure 2.1. An overview of a typical ERLANG/EFSM specification where i and j represent the arity of functions and events.

```
        [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 $0, \ldots, 8$, the value of `Stamina` is any integer in the range $0, \ldots, 2$ 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 declared for use by a tool e.g., to aid presentation of sequences of input and output events. For example, our tool, Erly Marsh see Section 6.7, uses a record to explicitly declare if input or output event, name of external node to communicate with, and associations between specific input and output event types (e.g., request and response).

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}`).

□

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 desribe 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,

```
-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()) -> external.
-spec progress(X::1..3) -> external.
-spec incident(I::bug | pernilla_call | kids_happy | kids_notok) -> external.
-spec checkout(Day::any()) -> external.
-spec checkin(Day::any()) -> external.

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.

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 $\texttt{\{}expr_1, \ldots, expr_n\texttt{\}}$ expresses a sequence of (ERLANG/EFSM) expressions separated by comma symbols and enclosed in braces. We sometimes also use $\overline{expr}$ for $expr_1, \ldots, expr_n$, $\overline{value}$ for $value_1, \ldots, value_n$ 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.

digit	::=	`0` | ... | `9`	
uppercase	::=	`A` | ... | `Z`	
lowercase	::=	`a` | ... | `z`	
digitletter	::=	*digit* | *uppercase* | *lowercase* | `_`	
integer(*i*)	::=	$digit^+$ | `-`$digit^+$	
unquotedatom	::=	*lowercase* $digitletter^*$	
quotedatom	::=	`'`$(char \mid escape)^+$`'`	
atom(*a*)	::=	*unquotedatom* | *quotedatom*	
bool	::=	`false` | `true`	
string	::=	`[`$char_1$`,` ... `,`$char_n$`]`	$n \geq 0$
u	::=	*uppercase* $digitletter^*$ | `_`	
arithmetic	::=	`+` | `-` | `*` | `div`	
relop	::=	`=/=` | `==` | `<` | `>` | `=<` | `>=`	
logop	::=	`and` | `not` | `or` | `xor`	
typetest	::=	`is_integer` | `is_boolean` | `is_record`	
	|	`is_tuple` | `is_list`	
guardop	::=	*relop* | *logop* | *typetest* | `size`	
fun `(`*f*`)`	::=	*atom*	
type `(`*t*`)`	::=	*atom*	
bv	::=	*atom* | *bool* | *integer* | `[]` | `{}`	
value(*d*)	::=	*bv* | `[`d_1`|`d_2`]` | `{`d_1`,` ... `,`d_n`}`	$n > 0$

Table 2.1. ERLANG *syntax base.*

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 ($\{d_1, \ldots, d_n\}$) and conses ($[d_1|d_2]$) where $d_1, \ldots, d_n$ 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 $[d_1, \ldots, d_n]$ is a shorthand for the list $[d_1|\ldots|[d_n|[]]\ldots]$.

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 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/Efsm 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/Efsm). In Erlang (and Erlang/Efsm) 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

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

Table 2.2. *A restricted set of the* Erlang *syntax.*

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 occuring 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/Efsm (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`,…,`e_n`}` and cons expression $[e_1|e_2]$ is evaluated by evaluating the sub-expressions e_1 to e_n. A function call expression f`(`e_1`,...,`e_n`)` is evaluated by evaluating the body of the function f with the results of the evaluation of the sub-expressions e_1 to e_n. 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_n 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*,{$a_1, \ldots, a_n$}).

In the record definition, *a* is the name of the record and $a_1, \ldots, a_n$ the names of the fields in the record. Given a record definition for *a*, a record can be created with #*a*{$a_1 = e_1, \ldots, a_{m_1} = e_{m_1}$}. A created record can (later) be updated with *e*#*a*{$a_1 = e_1, \ldots, a_{m_2} = e_{m_2}$} 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

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

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 $(cp_{k1}, \ldots, cp_{km})$ match the actual arguments, and the guard expression (g_k) evaluates to `true`, is evaluated.

Case and if expressions
The `case` expression

$$\begin{aligned} &\text{case } e \text{ of} \\ &\quad cp_1 \text{ when } g_1 \text{ -> } ce_1; \\ &\quad\quad \vdots \\ &\quad cp_n \text{ when } g_n \text{ -> } ce_n \\ &\text{end} \end{aligned}$$

is evaluated by first evaluating *e* to a value *d*, then matching *d* against patterns $cp_1, \ldots, cp_n$, 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

$$\texttt{if}\ \ g_1\ \texttt{->}\ ce_1;\ \cdots\ ;\ g_n\ \texttt{->}\ ce_n\ \ \texttt{end}$$

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

$$\begin{array}{l}\texttt{case true of}\\ \quad \texttt{\_ when}\ g_1\ \texttt{->}\ ce_1\texttt{;}\\ \quad\quad \vdots\\ \quad \texttt{\_ when}\ g_n\ \texttt{->}\ ce_n\\ \texttt{end}\end{array}$$

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

$$\texttt{-type (}t\texttt{() ::}\ tv_1\ \texttt{|...|}\ tv_n\texttt{).}$$

where t is a user defined type, previously not defined, with possible sub-types $tv_1, \ldots, tv_n$. 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`,...,`a_n`::`tv_n`}`.

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

$$\texttt{-spec}\ \ f\texttt{(}u_1\texttt{::}tv_1\texttt{,...,}u_n\texttt{::}tv_n\texttt{)}\ \texttt{->}\ tv_{out}\texttt{.}$$

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().
```

$$
\begin{array}{rcll}
v & ::= & \mathit{uppercase}\ \mathit{digitletter}^* & \\
 & & & \\
\mathit{atlist} & ::= & \texttt{[}\mathit{atom}_1\texttt{,}\ldots\texttt{,}\mathit{atom}_n\texttt{]} & n \geq 0 \\
\mathit{locdec} & ::= & \texttt{-locations(\{}\mathit{atom}\texttt{,}\mathit{atlist}_1\texttt{,}\mathit{atlist}_2\texttt{\}).} & \\
\mathit{statevardec} & ::= & \texttt{-statevars([}v_1,\ldots,v_n\texttt{]).} & \\
\mathit{configdec} & ::= & \texttt{-configs([}f_1\texttt{/}\mathit{integer}_1\texttt{,}\ldots\texttt{,}f_n\texttt{/}\mathit{integer}_n\texttt{]).} & n \geq 0 \\
\mathit{abstrscope} & ::= & \texttt{all} \mid \texttt{\{}\mathit{integer}_1\texttt{,}\mathit{integer}_2\texttt{\}} \mid \texttt{\{}f\texttt{,}\mathit{integer}\texttt{\}} & \\
\mathit{abstrdec} & ::= & \texttt{-abstr(\{}\mathit{abstrscope}\texttt{,}\mathit{string}_1\texttt{,}\mathit{string}_2\texttt{\}).} & \\
 & & & \\
\mathit{specdec} & ::= & \mathit{recdec} \mid \mathit{vtypedec} \mid \mathit{ftypedec} & \\
 & & \mid \mathit{statevardec} \mid \mathit{configdec} \mid \mathit{abstrdec} & \\
 & & & \\
\mathit{declarations} & ::= & \mathit{locdec}\ \mathit{specdec}^* & \\
\mathit{spec} & ::= & \mathit{declarations}\ \mathit{fundef}^+ &
\end{array}
$$

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

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()`.

□

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.

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 refering to a local variable and use v when refering 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 expresion 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 $0, \ldots, 9$.

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([`f_1`/`*integer*$_1$`, ... ,`f_n`/`*integer*$_n$`]).`

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 $f(e_1, \ldots, e_m)$ where f is the input event type and $e_1, \ldots, e_m$ 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 $f(e_1, \ldots, e_m)$ where f is the output event type and $e_1, \ldots, e_m$ 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. Ouput 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 occurence of an input event with some arity m. A set of transition clauses is of form

$$
\begin{aligned}
&l(a_0, u_1, \ldots, u_m) \texttt{ when } g_1 \texttt{ -> } ce_1\texttt{;}\\
&\qquad\vdots\\
&l(a_0, u_1, \ldots, u_m) \texttt{ when } g_n \texttt{ -> } ce_n\texttt{.}
\end{aligned}
$$

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

- a_0 is the input event type,
- $u_1, \ldots, u_m$ 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

  `{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 $u_1, \ldots, u_m$ 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 different input event types a with the same arity m, at the same 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., `{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 2.3.2 and Section 2.10.

2.3.3 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 $A >= 10$ 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 $A >= 10$ can be replaced by a boolean variable. Thus, the number of possible abstract test cases resulting from $A >= 10$ 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 when input event parameters have unexpected values. If it can be expected the behavior of the SUT is similar regardless on 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*`,`*string*$_1$`,`*string*$_2$`).`

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 `{`*integer*$_1$`,`*integer*$_2$`}`, 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/Efsm

The semantics for Erlang/Efsm 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/Efsm 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 2.1 A *structural state* is a tuple

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

where

- e is an Erlang/Efsm expression,
- $\overline{b}$ is a sequence of output events, each of which is of form $b(d_1, \ldots, d_n)$, 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

- $\rho \in U \to D$ is a local environment with bindings to the local variables U.

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

The evaluation of a ERLANG/EFSM expressions depends not only on the expression and the current global and local environments, but also on the actual definitions of the configuration access functions 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

$$\Delta \quad \models \qquad \langle e, \sigma, \rho \rangle \Rightarrow \left\langle d, \overline{b}, \sigma', \rho' \right\rangle,$$

denoting that e can be evaluated in a global environment σ, local environment ρ, and configuration environment Δ, to a value d. We always asssume 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, $\overline{b}$ 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 $\langle \sigma, \rho \rangle \models g$ denote that

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

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, $\langle \sigma, \rho \rangle \not\models g$ 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 $\langle e, \sigma, \rho \rangle \Rightarrow \left\langle d, \overline{b}, \sigma', \rho' \right\rangle$ exists and is of form:

$$\frac{cond_1, \ldots, cond_m \quad trans_1, \ldots, trans_n}{\langle e, \sigma, \rho \rangle \Rightarrow \left\langle d, \overline{b}, \sigma', \rho' \right\rangle}$$

where $cond_1, \ldots, cond_m$ are conditions under which the transition rule can be applied. This rule says that a transition rule is *enabled* for an ERLANG/EFSM expression e when the conditions $cond_1, \ldots, cond_m$ are satisfied and the transitions $trans_1, \ldots, trans_n$ 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/EFSM (and ERLANG), pattern matching is used to enable expressions to be evaluated and to bind values to local variables. We define a relation $match\,(cp, d, \rho, \rho')$ 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.

$$\frac{d \text{ is a basic or compound value}}{match\,(d, d, \rho, \rho)}$$

$$\frac{u \text{ is an anonymous local variable}}{match\,(u, d, \rho, \rho)}$$

$$\frac{u \text{ is a local variable, defined in } \rho}{match\,(u, \rho(u), \rho, \rho)}$$

$$\frac{u \text{ is a local variable, not defined in } \rho}{match\,(u, d, \rho, \rho[u \mapsto d])}$$

$$\frac{match\,(cp_1, d_1, \rho_0, \rho_1) \quad \ldots \quad match\,(cp_n, d_n, \rho_{n-1}, \rho_n)}{match\,(\{cp_1, \ldots, cp_n\}, \{d_1, \ldots, d_n\}, \rho_0, \rho_n)}$$

$$\frac{match\,(cp_1, d_1, \rho_0, \rho_1) \quad match\,(cp_2, d_2, \rho_1, \rho_2)}{match\,([cp_1|cp_2], [d_1|d_2], \rho_0, \rho_2)}$$

$$\frac{match\,(cp_1, d_1, \rho_0, \rho_1) \quad \ldots \quad match\,(cp_n, d_n, \rho_{n-1}, \rho_n)}{match\,(\overline{cp}, \overline{d}, \rho_0, \rho_n)}$$

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.

2.4.2 Transition Rules for ERLANG/EFSM Expressions

Basic and Compond values

All expressions in ERLANG/EFSM 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 evaluate to themselves.

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

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

$$\frac{\langle e_1, \sigma_0, \rho_0\rangle \Rightarrow \langle d_1, \overline{b_1}, \sigma_1, \rho_1\rangle \quad \ldots \quad \langle e_n, \sigma_{n-1}, \rho_{n-1}\rangle \Rightarrow \langle d_n, \overline{b_n}, \sigma_n, \rho_n\rangle}{\langle \{e_1, \ldots, e_n\}, \sigma_0, \rho_0\rangle \Rightarrow \langle [\{d_1, \ldots, d_n\}, \overline{b_1} \ldots \overline{b_n}, \sigma_n, \rho_n\rangle}$$

$$\frac{\langle e_1, \sigma_0, \rho_0\rangle \Rightarrow \langle d_1, \overline{b_1}, \sigma_1, \rho_1\rangle \quad \langle e_2, \sigma_1, \rho_1\rangle \Rightarrow \langle d_2, \overline{b_2}, \sigma_2, \rho_2\rangle}{\langle [e_1|e_2], \sigma_0, \rho_0\rangle \Rightarrow \langle [d_1|d_2], \overline{b_1 b_2}, \sigma_2, \rho_2\rangle}$$

In ERLANG (and ERLANG/EFSM) 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/EFSM is single assignment.

$$\frac{\begin{array}{c}\langle e, \sigma, \rho\rangle \Rightarrow \langle d, \overline{b}, \sigma', \rho''\rangle \\ match\,(cp, d, \rho'', \rho')\end{array}}{\langle cp = e, \sigma, \rho\rangle \Rightarrow \langle d, \overline{b}, \sigma', \rho'\rangle}$$

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.

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

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.

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

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

$$\frac{\begin{array}{rcl}\langle e_1, \sigma_0, \rho_0\rangle & \Rightarrow & \langle bool_1, \overline{b_1}, \sigma_1, \rho_1\rangle \\ & \vdots & \\ \langle e_n, \sigma_{n-1}, \rho_{n-1}\rangle & \Rightarrow & \langle bool_n, \overline{b_n}, \sigma_n, \rho_n\rangle\end{array}}{\langle \texttt{ or } (e_1, \ldots, e_n), \sigma_0, \rho_0\rangle \Rightarrow \langle \texttt{ or } (bool_1, \ldots, bool_n), \overline{b_1} \ldots \overline{b_n}, \sigma_n, \rho_n\rangle}$$

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/Efsm expression is sometimes written with parenthesis in the following transition rules.

$$\frac{\langle ce_0, \sigma, \rho\rangle \Rightarrow \langle d, \overline{b_0}, \sigma'', \rho''\rangle \quad \langle ce_1, \sigma'', \rho''\rangle \Rightarrow \langle d', \overline{b_1}, \sigma', \rho'\rangle}{\langle (ce_0 \texttt{,} ce_1), \sigma, \rho\rangle \Rightarrow \langle d', \overline{b_0 b_1}, \sigma', \rho'\rangle}$$

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`.

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

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 evalutes 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.

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

User Defined Functions

A call to a user defined function $f(e_1, \ldots, e_m)$, defined by the Erlang/Efsm clauses

$$\begin{array}{l} f\texttt{(}cp_{11}\texttt{,} \ldots \texttt{,}cp_{1m}\texttt{)} \texttt{ when } g_1 \texttt{ -> } ce_1\texttt{;} \\ \quad \vdots \\ f\texttt{(}cp_{n1}\texttt{,} \ldots \texttt{,}cp_{nm}\texttt{)} \texttt{ when } g_n \texttt{ -> } ce_n\texttt{.} \end{array}$$

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

$$\frac{\begin{array}{c}\forall k : 1 \leq k \leq m : \langle e_k, \sigma_{k-1}, \rho_{k-1}\rangle \Rightarrow \langle d_k, \overline{b_k}, \sigma_k, \rho_k\rangle \\ \forall j, 1 \leq j < i : match\left(\overline{cp}_j, \overline{d}, \bullet, \rho_t\right) \text{ implies } \langle \sigma_m, \rho_t\rangle \not\models g_j \\ match\left(\overline{cp}_i, \overline{d}, \bullet, \rho'\right) \\ \langle \sigma_m, \rho'\rangle \models g_i \\ \langle ce_i, \sigma_m, \rho'\rangle \Rightarrow \langle d, \overline{b}, \sigma', \rho''\rangle\end{array}}{\langle f(e_1, \ldots, e_m), \sigma_0, \rho_0\rangle \Rightarrow \langle d, \overline{b_1} \cdots \overline{b_m}\ \overline{b}, \sigma', \rho''\rangle}$$

Configuration access functions
A call to a configuration access function $f(e_1, \ldots, e_m)$ where the name, f, is the name of a configuration access function is evaluated by evaluating all arguments $e_1, \ldots, e_m$ to $d_1, \ldots, d_m$, 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.

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

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

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

$$\frac{\begin{array}{rcl} \langle e_1, \sigma_0, \rho_0\rangle & \Rightarrow & \langle d_1, \overline{b_1}, \sigma_1, \rho_1\rangle \\ & \vdots & \\ f \in A_b \qquad \langle e_m, \sigma_{m-1}, \rho_{m-1}\rangle & \Rightarrow & \langle d_m, \overline{b_m}, \sigma_m, \rho_m\rangle \end{array}}{\langle f(e_1, \ldots, e_m), \sigma_0, \rho_0\rangle \Rightarrow \langle \mathtt{true}, \overline{b_1} \cdots \overline{b_m}\ f(d_1, \ldots, d_m), \sigma_m, \rho_m\rangle}$$

2.5 Derived State Machines

Usin the semantics of Erlang/Efsm, defined in Section 2.4, we can now describe how a Erlang/Efsm 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/Efsm 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

$$\begin{aligned} &l\texttt{(}a_0\texttt{,}u_1\texttt{,...,}u_m\texttt{)}\ \texttt{when}\ g_1\ \texttt{->}\ ce_1\texttt{;} \\ &\qquad\vdots \\ &l\texttt{(}a_0\texttt{,}u_1\texttt{,...,}u_m\texttt{)}\ \texttt{when}\ g_n\ \texttt{->}\ ce_n\texttt{.} \end{aligned}$$

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

$$\Delta \quad \models \quad \langle l(a\texttt{,}d_1\texttt{,...,}d_m), \sigma, \bullet \rangle \Rightarrow \langle \texttt{\{next\_state,}l'\texttt{\}}, \overline{b}, \sigma', \rho \rangle \ ,$$

i.e., acccording to the rule for user defined functions,
$l\texttt{(}a_0\texttt{,}u_1\texttt{,...,}u_m\texttt{)}\ \texttt{when}\ g_i\ \texttt{->}\ ce_i$ 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 $u_1\texttt{,...,}u_m$ are bound to to $d_1, \ldots, d_m$, and the corresponding clause expression ce_i evaluates to $\texttt{\{next\_state,}l'\texttt{\}}$. Then there is a transition from the state $\langle l, \sigma \rangle$ to the state $\langle l', \sigma' \rangle$, triggered by the input event $a(d_1, \ldots, d_m)$, and emitting the sequence $\overline{b}$ 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

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

Each computation step $\langle l, \sigma \rangle \xrightarrow{a(\overline{d})/\overline{b}} \langle l', \sigma' \rangle$ 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/EFSM 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

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

which starts in the initial state.

A *trace* is the finite sequence

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

of labels of such a run. Each label is a pair consisting of an input event and sequence of output events. A *run over a trace* $a_1(\overline{d_1})/\overline{b_1} \ \cdots \ a_n(\overline{d_n})/\overline{b_n}$ is a run whose sequence of labels is exactly this trace.

A *test case* is a pair

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

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

$$\texttt{is\_tuple}(p_2)\ \texttt{and}\ \texttt{size}(p_2) == 3,$$

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/EFSM *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 $\mathfrak{e}$ to range over normal form expressions. Let $\mathfrak{E}$ be the domain of ERLANG/EFSM normal form expressions. A *symbolic parameter* is either

- an *input expression parameter*, i.e., a parameter p_i in an input expression $a(p_1, \ldots, p_n)$ where a is an input event type, or
- a *configuration parameter*, i.e., a configuration access function of form $f(\mathfrak{e}_1, \ldots \mathfrak{e}_n)$ with ERLANG/EFSM normal form expressions $\mathfrak{e}_1, \ldots \mathfrak{e}_n$ as arguments.

Note that a configuration parameter may depend on input expression parameters. Thus, we may have different constant configuration data for each possible evaluation $d_1, \ldots, d_n$ of $\mathfrak{e}_1, \ldots \mathfrak{e}_n$.

Definition 2.2 A *symbolic structural state* is a tuple

$$\langle e, g, \overline{\mathfrak{b}}, \sigma, \rho \rangle$$

where

- e is an ERLANG/EFSM expression,
- g is an ERLANG/EFSM guard expression,
- $\overline{\mathfrak{b}}$ is a sequence of output expressions, each of which is of form

$$b(\mathfrak{e}_1, \ldots, \mathfrak{e}_n),$$

  such that $b \in A_b$ is an output event type and $\mathfrak{e}_1, \ldots, \mathfrak{e}_n$ are ERLANG/EFSM normal form expressions where each $\mathfrak{e}_j \in \mathfrak{E}$,
- $\sigma \in V \to \mathfrak{E}$ is a global symbolic environment with bindings to the global state variables V, and
- $\rho \in U \to \mathfrak{E}$ is a local symbolic environment with bindings to the local variables U.

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

$$\langle a(\overline{p}), \texttt{true}, \epsilon, \bullet, \bullet \rangle$$

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

We always asssume 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 $\langle e, \texttt{true}, \epsilon, \sigma, \rho \rangle$, as $\langle e, \sigma, \rho \rangle$.

A *symbolic transition* between two structural states

$$\langle e, \sigma, \rho \rangle \Rightarrow \langle \mathfrak{e}, g, \overline{\mathfrak{b}}, \sigma', \rho' \rangle$$

denotes that the expression e is evaluated to an ERLANG/EFSM normal form expression $\mathfrak{e}$, in a global symbolic environment σ and local symbolic environment ρ, under the condition g. The evaluation generates the sequence $\overline{\mathfrak{b}}$ 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:

$$\frac{cond_1, \ldots, cond_m \quad trans_1, \ldots, trans_n}{\langle e, \sigma, \rho \rangle \Rightarrow \langle \mathfrak{e}, g, \overline{\mathfrak{b}}, \sigma', \rho' \rangle}$$

where $cond_1, \ldots, cond_m$ are conditions on when the symbolic transition rule can be applied, and $trans_1, \ldots, trans_n$ are symbolic transitions on which the symbolic transition rule depends. A symbolic transition rule is *enabled* for an Erlang/Efsm 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 $match\,(cp, \mathfrak{e}, \rho, g', \rho')$ in Figure 2.4 to denote a successful pattern match between a pattern cp and an Erlang/Efsm normal form expression $\mathfrak{e}$ 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 $\mathfrak{e}$. The definition is a straightforward modification of $match\,(cp, d, \rho, \rho')$ in Figure 2.3 where we additional 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/Efsm Expressions

Basic and Compond values

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

$$\frac{d \text{ is a basic or compound value}}{match\,(d, d, \rho, \texttt{true}, \rho)}$$

$$\frac{u \text{ is an anonymous local variable}}{match\,(u, \mathfrak{e}, \rho, \texttt{true}, \rho)}$$

$$\frac{u \text{ is a local variable, defined in } \rho}{match\,(u, \mathfrak{e}, \rho, \rho(u) == \mathfrak{e}, \rho)}$$

$$\frac{u \text{ is a local variable, not defined in } \rho}{match\,(u, \mathfrak{e}, \rho, \texttt{true}, \rho[u \mapsto \mathfrak{e}])}$$

$$\frac{\begin{array}{c} match\,(cp_1, \texttt{element(}1, \mathfrak{e}\texttt{)}, \rho_0, g_1, \rho_1) \\ \vdots \\ match\,(cp_n, \texttt{element(}n, \mathfrak{e}\texttt{)}, \rho_{n-1}, g_n, \rho_n) \end{array}}{match\left(\texttt{\{}cp_1, \ldots, cp_n\texttt{\}}, \mathfrak{e}, \rho_0, \left(\begin{array}{ll} & \texttt{is\_tuple(}\mathfrak{e}\texttt{)} \\ \wedge & \texttt{size(}\mathfrak{e}\texttt{)} == n \\ \wedge & g_1 \wedge \ldots \wedge g_n \end{array}\right), \rho_n\right)}$$

$$\frac{match\,(cp_1, \texttt{hd(}\mathfrak{e}_1\texttt{)}, \rho_0, g_1, \rho_1) \quad match\,(cp_2, \texttt{tail(}\mathfrak{e}_2\texttt{)}, \rho_1, g_n, \rho_2)}{match\,([cp_1|cp_2], \mathfrak{e}, \rho_0, \texttt{is\_list(}\mathfrak{e}\texttt{)} \wedge g_1 \wedge g_2, \rho_2)}$$

$$\frac{match\,(cp_1, \mathfrak{e}_1, \rho_0, g_1, \rho_1) \quad \ldots \quad match\,(cp_n, \mathfrak{e}_n, \rho_{n-1}, g_n, \rho_n)}{match\,(\overline{cp}, \overline{\mathfrak{e}}, \rho_0, g_1 \wedge \ldots \wedge g_n, \rho_n)}$$

Figure 2.4. Definition of the symbolic $match\,(cp, \mathfrak{e}, \rho, g', \rho')$ relation for a pattern match between a pattern cp and an ERLANG/EFSM normal form expression $\mathfrak{e}$. 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.

first evaluating all sub-expressions in a left-to-right order.

$$\frac{\begin{array}{c}\langle e_1, \sigma_0, \rho_0\rangle \Rightarrow \langle \mathfrak{e}_1, g_1, \overline{\mathfrak{b}_1}, \sigma_1, \rho_1\rangle \\ \vdots \\ \langle e_n, \sigma_{n-1}, \rho_{n-1}\rangle \Rightarrow \langle \mathfrak{e}_n, g_n, \overline{\mathfrak{b}_n}, \sigma_n, \rho_n\rangle\end{array}}{\langle \{e_1, \ldots, e_n\}, \sigma_0, \rho_0\rangle \Rightarrow \langle \{\mathfrak{e}_1, \ldots, \mathfrak{e}_n\}, g_1 \wedge \ldots \wedge g_n, \overline{\mathfrak{b}_0} \ldots \overline{\mathfrak{b}_n}, \sigma_n, \rho_n\rangle}$$

$$\frac{\langle e_1, \sigma_0, \rho_0\rangle \Rightarrow \langle \mathfrak{e}_1, g_1, \overline{\mathfrak{b}_1}, \sigma_1, \rho_1\rangle \quad \langle e_2, \sigma_1, \rho_1\rangle \Rightarrow \langle \mathfrak{e}_2, g_2, \overline{\mathfrak{b}_2}, \sigma_2, \rho_2\rangle}{\langle [e_1|e_2], \sigma_0, \rho_0\rangle \Rightarrow \langle [\mathfrak{e}_1|\mathfrak{e}_2], g_1 \wedge g_2, \overline{\mathfrak{b}_1\mathfrak{b}_2}, \sigma_2, \rho_2\rangle}$$

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/Efsm normal form expressions by pattern matching.

$$\frac{\begin{array}{c}\langle e, \sigma, \rho\rangle \Rightarrow \langle \mathfrak{e}, g, \overline{\mathfrak{b}}, \sigma', \rho''\rangle \\ match\,(cp, \mathfrak{e}, \rho'', g', \rho')\end{array}}{\langle cp = e, \sigma, \rho\rangle \Rightarrow \langle \mathfrak{e}, g \wedge g', \overline{\mathfrak{b}}, \sigma', \rho'\rangle}$$

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

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

All variables must be bound to an Erlang/Efsm 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.

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

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

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

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

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

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

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 $1, \ldots, i-1$ 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.

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

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 evalutes 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/Efsm 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.

$$\frac{\begin{array}{l}\langle e, \sigma, \rho\rangle \Rightarrow \langle \mathfrak{e}^e, g^e, \overline{\mathfrak{b}_1}, \sigma', \rho'\rangle \\ 1 \leq i \leq n \\ \forall j \text{ such that } 1 \leq j < i : \\ \quad \text{if there are no } g'_j,\ \rho''_j \text{ s.t. } match\left(cp_j, \mathfrak{e}^e, \rho', g'_j, \rho''_j\right) \\ \quad\quad \text{then let } g'_j,\ g''_j, \text{ be } \texttt{false} \\ \quad \text{else let } g'_j,\ \rho''_j,\ g''_j \text{ be s.t.} \\ \quad\quad match\left(cp_j, \mathfrak{e}^e, \rho', g'_j, \rho''_j\right) \text{ and } \langle g_j, \sigma', \rho''_j\rangle \Rightarrow \langle g''_j, \texttt{true}, \epsilon, \sigma', \rho''_j\rangle \\ match\,(cp_i, \mathfrak{e}^e, \rho', g'_i, \rho''_i) \\ \langle g_i, \sigma', \rho''_i\rangle \Rightarrow \langle g''_i, \texttt{true}, \epsilon, \sigma', \rho''_i\rangle \\ \langle ce_i, \sigma', \rho''_i\rangle \Rightarrow \langle \mathfrak{e}, g''', \overline{\mathfrak{b}_2}, \sigma'', \rho'''\rangle \\ g = g^e \wedge \neg(g'_1 \wedge g''_1) \wedge \ldots \wedge \neg(g'_{i-1} \wedge g''_{i-1}) \wedge (g'_i \wedge g''_i) \wedge g'''\end{array}}{\left\langle \left(\begin{array}{l}\texttt{case } e \texttt{ of} \\ \quad cp_1 \texttt{ when } g_1 \texttt{ -> } ce_1\texttt{;} \\ \quad\quad \vdots \\ \quad cp_n \texttt{ when } g_n \texttt{ -> } ce_n \\ \texttt{end}\end{array}\right), \sigma, \rho \right\rangle \Rightarrow \langle \mathfrak{e}, g, \overline{\mathfrak{b}_1 \mathfrak{b}_2}, \sigma'', \rho'''\rangle}$$

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

Clause Selection in User Defined Functions

A call to a user defined function $f(e_1, \ldots, e_m)$, defined by the Erlang/Efsm clauses

$$\begin{array}{l} f(cp_{11}, \ldots, cp_{1m}) \texttt{ when } g_1 \texttt{ -> } ce_1\texttt{;} \\ \quad \vdots \\ f(cp_{n1}, \ldots, cp_{nm}) \texttt{ when } g_n \texttt{ -> } ce_n\texttt{.} \end{array}$$

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.

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

Configuration Access Functions

A call to a configuration access function $f(e_1, \ldots, e_m)$ to access constant configuration data, is evaluated simply by evaluating all arguments $e_1, \ldots, e_m$ to $\mathfrak{e}_1, \ldots, \mathfrak{e}_m$. 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.

$$\frac{\begin{array}{c}\langle e_1, \sigma_0, \rho_0\rangle \Rightarrow \langle \mathfrak{e}_1, g_1, \overline{\mathfrak{b}_1}, \sigma_1, \rho_1\rangle \\ \vdots \\ f \in C \qquad \langle e_m, \sigma_{m-1}, \rho_{m-1}\rangle \Rightarrow \langle \mathfrak{e}_m, g_m, \overline{\mathfrak{b}_m}, \sigma_m, \rho_m\rangle\end{array}}{\langle f(e_1, \ldots e_m), \sigma_0, \rho_0\rangle \Rightarrow \langle f(\mathfrak{e}_1, \ldots \mathfrak{e}_m), g_1 \wedge \ldots \wedge g_m, \overline{\mathfrak{b}_1} \cdots \overline{\mathfrak{b}_m}, \sigma_m, \rho_m\rangle}$$

Output Expressions

A call to an output expression $f(e_1, \ldots, e_m)$ where the name, f, is an output event type, is evaluated by first evaluating all arguments $e_1, \ldots, e_m$ to $\mathfrak{e}_1, \ldots, \mathfrak{e}_m$.

$$\frac{\begin{array}{c}\langle e_1, \sigma_0, \rho_0\rangle \Rightarrow \langle \mathfrak{e}_1, g_1, \overline{\mathfrak{b}_1}, \sigma_1, \rho_1\rangle \\ \vdots \\ f \in A_b \qquad \langle e_m, \sigma_{m-1}, \rho_{m-1}\rangle \Rightarrow \langle \mathfrak{e}_m, g_m, \overline{\mathfrak{b}_m}, \sigma_m, \rho_m\rangle\end{array}}{\langle f(e_1, \ldots, e_m), \sigma_0, \rho_0\rangle \Rightarrow \langle \texttt{true}, g, \overline{\mathfrak{b}_1} \cdots \overline{\mathfrak{b}_m}\ f(\mathfrak{e}_1, \ldots \mathfrak{e}_m), \sigma_m, \rho_m\rangle}$$

where $g = g_1 \wedge \ldots \wedge g_m$,.

2.8 Correspondence Between non-Symbolic and Symbolic Semantics

In the previous section, the relationship between the symbolic semantics and the original operational semantics was described informally. In this section, we make the relation more precise.

Recall that the (non-symbolic) operational semantics defines transitions of the form

$$\langle e, \sigma, \rho \rangle \Rightarrow \left\langle d, \overline{b}, \sigma', \rho' \right\rangle,$$

while the symbolic operational semantics defines transitions of the form

$$\langle e, \sigma, \rho \rangle \Rightarrow \left\langle \mathfrak{e}, g, \overline{\mathfrak{b}}, \sigma', \rho' \right\rangle,$$

where the environments σ and ρ (and σ' and ρ') 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 configuration access functions).

In order to relate normal form expressions to values, we must introduce mappings from symbolic parameters to values. So, define a *symbolic parameter* to be either an input expression parameter p, or the application $f(\mathfrak{e}_1, \ldots \mathfrak{e}_n)$ of a configuration access function f to a tuple of normal form expressions $\mathfrak{e}_1, \ldots \mathfrak{e}_n$. Define a *symbolic environment* Γ to be a mapping from symbolic parameters to values. For a normal form expression $\mathfrak{e}$, we write $\Gamma(\mathfrak{e})$ for the value obtained by replacing each symbolic parameter p in $\Gamma(\mathfrak{e})$ by $\Gamma(p)$, and evaluating the result. For a symbolic environment ρ, we write $\Gamma(\rho)$ for the environment ρ' such that $\rho'(u) = \Gamma(\rho(u))$. A symbolic environment Γ must satisfy the technical assumption that if $\Gamma(\mathfrak{e}_i) = \Gamma(\mathfrak{e}'_i)$ for $i = 1, \ldots, n$ then $\Gamma(f(\mathfrak{e}_1, \ldots \mathfrak{e}_n)) = \Gamma(f(\mathfrak{e}'_1, \ldots \mathfrak{e}'_n))$ whenever f is a configuration access function.

The relationship between concrete and symbolic operational semantics can now be formulated as in the following proposition.

Proposition 2.1 A symbolic transition of form

$$\langle e, \sigma, \rho \rangle \Rightarrow \left\langle \mathfrak{e}, g, \overline{\mathfrak{b}}, \sigma', \rho' \right\rangle$$

denotes that for each symbolic environment Γ such that $\Gamma(g)$ is true, we have

$$\langle e, \Gamma(\sigma), \Gamma(\rho) \rangle \Rightarrow \left\langle \Gamma(\mathfrak{e}), \Gamma(\overline{\mathfrak{b}}), \Gamma(\sigma'), \Gamma(\rho') \right\rangle.$$

Furthermore, whenever $\langle e, \Gamma(\sigma), \Gamma(\rho) \rangle \Rightarrow \left\langle d, \overline{b}, \sigma'', \rho'' \right\rangle$ for some (non-symbolic) environments σ'', ρ'', then there is a symbolic transition of form $\langle e, \sigma, \rho \rangle \Rightarrow \left\langle \mathfrak{e}, g, \overline{\mathfrak{b}}, \sigma', \rho' \right\rangle$. □

In order to establish Proposition 2.1, we must establish a corresponding relationship between concrete and symbolic operational semantics for each construct considered by the semantics. Let us go through such constructs one by one.

Pattern Matching
The relation $match\,(cp, \mathfrak{e}, \rho, g', \rho')$ denotes that for each symbolic environment Γ such that $\Gamma(g')$ holds, we have $match\,(cp, \Gamma(\mathfrak{e}), \Gamma(\rho), \Gamma(\rho'))$. This relationship can be checked by comparing each rule in Figure 2.4 with the corresponding rule in Figure 2.3.

Basic and Compound values
That Proposition 2.1 holds when e is a tuple or cons, follows directly. Let us for example consider the case when e is a cons $[e_1|e_2]$. Consider a symbolic structural state $\langle [e_1|e_2], \sigma_0, \rho_0\rangle$ and a symbolic environment Γ such that $\Gamma(g_1 \wedge g_2)$ holds. By the rule for conses in the symbolic semantics, we have

$$\langle e_1, \sigma_0, \rho_0\rangle \Rightarrow \langle \mathfrak{e}_1, g_1, \overline{\mathfrak{b}_1}, \sigma_1, \rho_1\rangle \quad \text{and} \quad \langle e_2, \sigma_1, \rho_1\rangle \Rightarrow \langle \mathfrak{e}_2, g_2, \overline{\mathfrak{b}_2}, \sigma_2, \rho_2\rangle,$$

implying by Proposition 2.1, and a suitable induction hypothesis, that

$$\langle e_1, \Gamma(\sigma_0), \Gamma(\rho_0)\rangle \Rightarrow \langle \Gamma(\mathfrak{e}_1), \Gamma(\overline{\mathfrak{b}_1}), \Gamma(\sigma_1), \Gamma(\rho_1)\rangle$$

and

$$\langle e_2, \Gamma(\sigma_1), \Gamma(\rho_1)\rangle \Rightarrow \langle \Gamma(\mathfrak{e}_2), \Gamma(\overline{\mathfrak{b}_2}), \Gamma(\sigma_2), \Gamma(\rho_2)\rangle.$$

By the non-symbolic semantics, we then have

$$\langle [e_1|e_2], \Gamma(\sigma_0), \Gamma(\rho_0)\rangle \Rightarrow \langle \Gamma([\mathfrak{e}_1|\mathfrak{e}_2]), \Gamma(\overline{\mathfrak{b}_1\mathfrak{b}_2}), \Gamma(\sigma_2), \Gamma(\rho_2)\rangle.$$

Similar reasoning can be used for Local and State variables and Built-in functions. For If expressions, we note that the symbolic evaluation of a guard will not generate any path condition.

If there are g_1' and ρ_1'' such that $match\,(cp_1, \mathfrak{e}^e, \rho', g_1', \rho_1'')$, then let g_1'' and ρ_1'' be such that $\langle g_1, \sigma', \rho_1''\rangle \Rightarrow \langle g_1'', \texttt{true}, \epsilon, \sigma', \rho_1''\rangle$. If there are no such g_1' and ρ_1'', then let g_1' be `false`.

This approach works if we assume that a non-taken clause leaves no binding in the environment, and that in the symbolic semantics, there is no "branching" in the evaluation of a pattern match.

2.9 Deriving a symbolic representation of State Machines

We can now define a symbolic version of the computation step-based semantics defined by an Erlang/Efsm 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 $\langle l_0, \bullet, \texttt{true} \rangle$ where l_0 is the initial location in the EFSM and $\bullet$ is the empty environment.

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

$$\langle l, \sigma, G \rangle \overset{a(\overline{p})/\overline{\mathfrak{b}}}{\rightarrow} \langle l', \sigma', G' \rangle \quad ,$$

where

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

such that for location l and input expression $a(p_1, \ldots, p_n)$, the transition rules for user defined functions, considering l as a user defined function, allow to derive the symbolic transition

$$\langle l(a\,, p_1\,, \ldots\,, p_n), \sigma, \bullet \rangle \Rightarrow \langle \{\texttt{next\_state}\,, l'\}, g, \overline{\mathfrak{b}}, \sigma', \rho \rangle$$

where G' is $G \ \wedge \ g$.

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

$$\langle l_0, \bullet, \texttt{true} \rangle \overset{a_1(\overline{p_1})/\overline{\mathfrak{b}_1}}{\longrightarrow} \langle l_1, \sigma_1, G_1 \rangle \overset{a_2(\overline{p_2})/\overline{\mathfrak{b}_2}}{\longrightarrow} \ \cdots \ \overset{a_n(\overline{p_n})/\overline{\mathfrak{b}_n}}{\longrightarrow} \langle l_n, \sigma_n, G_n \rangle$$

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

$$a_1(\overline{p_1})/\overline{\mathfrak{b}_1} \quad a_2(\overline{p_2})/\overline{\mathfrak{b}_2} \quad \cdots \quad a_n(\overline{p_n})/\overline{\mathfrak{b}_n}$$

of labels that occur in a symbolic run of the EFSM. Each label consists of an input expression and a sequence of output expressions. 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 2.3 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. □

2.10 Defining and Normalizing Edge Clauses

A Erlang/Efsm 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 definitin 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/EFSM specification of Figure 2.2 (from the example in Section 2.2).

Whenever the symbolic semantics generates a transition of form

$$\langle l(a\texttt{,}p_1\texttt{,}\ldots\texttt{,}p_n), \sigma, \bullet\rangle \Rightarrow \langle \texttt{\{next\_state,}l'\texttt{\}}, g, \overline{\mathfrak{b}}, \sigma', \rho\rangle,$$

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

$$l\texttt{(}a_{k0}\texttt{,}u_{k1}\texttt{,}\ldots\texttt{,}u_{km}\texttt{) when }g'_k\texttt{ -> }\overline{v_k = \mathfrak{e}_k}\texttt{, }\overline{\mathfrak{b}_k}\texttt{, \{next\_state,}l'\texttt{\}}$$

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

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

$$\begin{gathered} l\texttt{(}a_{10}\texttt{,}u_{11}\texttt{,}\ldots\texttt{,}u_{1m}\texttt{) when }g_1\texttt{ -> }\overline{v_1 = \mathfrak{e}_1}\texttt{, }\overline{\mathfrak{b}_1}\texttt{, \{next\_state,}l'\texttt{\}} \\ \vdots \\ l\texttt{(}a_{n0}\texttt{,}u_{n1}\texttt{,}\ldots\texttt{,}u_{nm}\texttt{) when }g_n\texttt{ -> }\overline{v_n = \mathfrak{e}_n}\texttt{, }\overline{\mathfrak{b}_n}\texttt{, \{next\_state,}l'\texttt{\}} \end{gathered}$$

representing edges from location l to location l' when triggered by an input event $a(d_1, \ldots, d_m)$. Now, if all (or any subset) of these edge clauses have the same bindings to state variables (i.e., $\overline{v_1 = \mathfrak{e}_1}, \ldots, \overline{v_n = \mathfrak{e}_n}$ are

all equal to $\overline{v = \mathfrak{e}}$), and the same output expressions (i.e., $\overline{\mathfrak{b}_1}, \ldots, \overline{\mathfrak{b}_n}$ are all equal to $\overline{\mathfrak{b}}$), they can be merged into a single edge clause, i.e.,

$$l\texttt{(}a\texttt{,}u_1\texttt{,}\ldots\texttt{,}u_m\texttt{) when } g_1 \vee, \ldots, \vee g_n \texttt{ -> } \overline{v = \mathfrak{e}}\texttt{, } \overline{\mathfrak{b}}\texttt{, \{next\_state,}l'\texttt{\}}$$

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 for the rewriting 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 [Eri 15], The rewriting 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.

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. Here we use the `state` record and define a 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.

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.

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.

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'}).
```

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. 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 2.11.1 and Section 2.11.1. 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.

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 was excluded. This is 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

```
-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.

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.

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

This chapter is organized as follows. Section 3.1 gives an introduction to observers, Section 3.2 gives a more detailed description of Erlang/Obs as a language for specifing observers, Section 3.3 describe how to extend Erlang/Obs with user defined observer predicates, Section 3.4 introduces a graphical notation for specifying observers, Section 3.5 describes the operational semantics for observers in Erlang/Obs, Section 3.6 explains how an observer is derived from an Erlang/Obs definition, Section 3.7 describes the corresponding operational semantics for an observers with symbolic execution, and Section 3.8 explains how to derive a symbolic observer. Section 3.8 puts things together by giving definitions for executing *sequences* of observer edges onto an EFSM.

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

specification, (cf. Section 2.5), 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

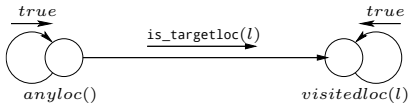


Figure 3.1. An observer representing the coverage item *"visit location l of the EFSM"* in the *visitedloc*(l) state.

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*

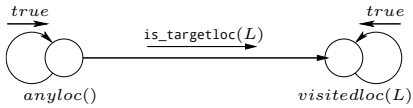


Figure 3.2. An observer representing the coverage item *"all target locations of the EFSM"* in the *visitedloc*(L) state.

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

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

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/Efsm *edge clause*. Thus, we define an *annotated computation step* as a pair

$$\left\langle \begin{array}{l} \langle l, \sigma \rangle \xrightarrow{a(\overline{d})/\overline{b}} \langle l', \sigma' \rangle, \\ l\texttt{(}a\texttt{,}u_1\texttt{,}\ldots\texttt{,}u_m\texttt{)}\ \texttt{when}\ g\ \texttt{->}\ \overline{v = \mathfrak{e}},\ \overline{\mathfrak{b}},\ \texttt{\{next\_state,}l'\texttt{\}} \end{array} \right\rangle$$

consisting of

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

$$l\texttt{(}a\texttt{,}u_1\texttt{,}\ldots\texttt{,}u_m\texttt{)}\ \texttt{when}\ g\ \texttt{->}\ \overline{v = \mathfrak{e}},\ \overline{\mathfrak{b}},\ \texttt{\{next\_state,}l'\texttt{\}}\ ,$$

  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 $\langle \Sigma, \mathcal{Q}, q_0, \mathcal{Q}_f, \rightarrow \rangle$ where

- Σ, the input alphabet for an observer, is the set of annotated computation steps of the Erlang/Efsm specification.
- $\mathcal{Q}$, ranged over by q, is a finite set of *observer states*. An observer state is of form $\iota\texttt{(}d_1\texttt{,}\ldots\texttt{,}d_n\texttt{)}$, where $\iota \in L^o$ is an observer location, and $d_1, \ldots, d_n \in D$ are values.

- q_0 is the *initial observer state*.
- $\mathcal{Q}_f \subseteq \mathcal{Q}$ is the set of *accepting observer states*, representing coverage items.
- $\rightarrow \subseteq \mathcal{Q} \times \Sigma \times \mathcal{Q}$ 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 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 $\langle l_0, \bullet \rangle \xrightarrow{a_1(\overline{d_1})/\overline{b_1}} \cdots \xrightarrow{a_n(\overline{d_n})/\overline{b_n}} \langle l_n, \sigma_n \rangle$ be a run of an EFSM, and let for $i = 1, \ldots, n$, ς_i represent the ith annotated computation step

$$\left\langle \begin{array}{l} \langle l_{i-1}, \sigma_{i-1} \rangle \xrightarrow{a_i(\overline{d_i})/\overline{b_i}} \langle l_i, \sigma_i \rangle, \\ l_{i-1}\texttt{(}a\texttt{,}u_1\texttt{,}\ldots\texttt{,}u_m\texttt{) when } g \texttt{ -> } \overline{v = \mathfrak{e}}\texttt{, } \overline{\mathfrak{b}}\texttt{, \{next\_state,}l_i\texttt{\}} \end{array} \right\rangle$$

Then we say that the run *covers* a coverage item $q_f \in \mathcal{Q}_f$ if there is a run

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

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.

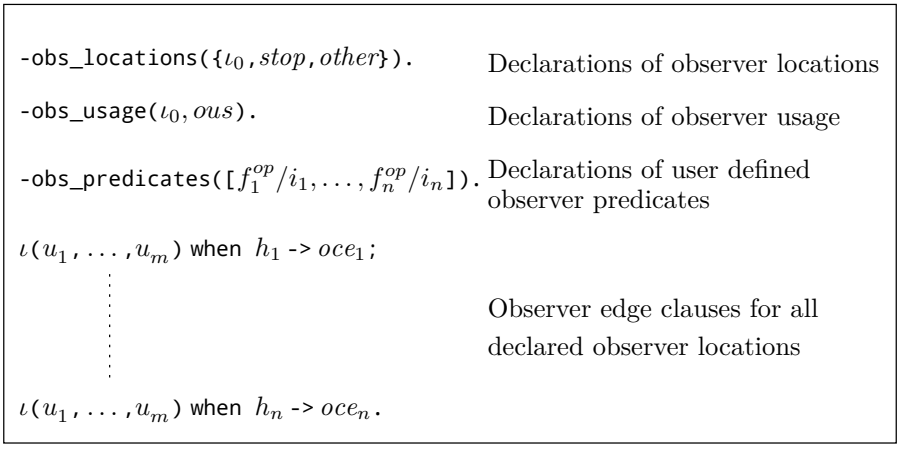


Figure 3.3. An overview of a typical Erlang/Obs specification.

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 exists 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/Efsm 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 consist 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 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 on 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, `is_edge(E)` will always succeed and bind `E` to a value. In `q1` 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`, `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`.

□

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 use 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`,…,`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 state* is of form ι`(`d_1`,...,`d_n`)` where ι is an *observer location*, and $d_1, \ldots, d_n$ 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

$$\iota\texttt{(}oe_1\texttt{,}\ldots\texttt{,}oe_n\texttt{)}$$

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

$emop$	::=	`is_targetloc` | `is_sourceloc` | `is_definedvar` | `is_usedvar`	
	|	`is_edge` | `is_startloc` | `is_stoploc` | `is_da`	
	|	`is_sourceval` | `is_targetval` | `is_cc` | `is_mcc` | `is_mcdc`	
	|	`is_eventtype` | `is_eventpars`	
f^{op}	::=	$emop \mid atom$	
ι	::=	$atom$	
ous	::=	`observer` | `filter` | `property`	
$obsge$	::=	$bv \mid u$	
	|	f^{op}`(`$obsge_1$`,` $\ldots$ `,`$obsge_n$`)`	$n \geq 0$
	|	$guardop$`(`$obsge_1$`,` $\ldots$ `,`$obsge_n$`)`	$n > 0$
h	::=	$obsge_1$`,` $\ldots$ `,`$obsge_n$	$n > 0$
oe	::=	$bv \mid u \mid$ `[`oe_1`|`oe_2`]` | `{`oe_1`,` $\ldots$ `,`oe_n`}` | `(`oe`)`	$n > 0$
	|	cp`=`oe	
	|	$guardop$`(`oe_1`,` $\ldots$ `,`oe_n`)`	$n > 0$
	|	$arithmetic$`(`oe_1`,` $\ldots$ `,`oe_n`)`	$n > 0$
	|	`{next_state,`ι`(`oe_1`,` $\ldots$ `,`oe_n`)}`	$n \geq 0$
oce	::=	oe_1`,` $\ldots$ `,`oe_n	$n > 0$
$obsedge$	::=	ι`(`u_1`,` $\ldots$ `,`u_m`)` `when` h_1 `->` oce_1`;` $\vdots$ ι`(`u_1`,` $\ldots$ `,`u_m`)` `when` h_n `->` oce_n`.`	$m \geq 0$ $n > 0$
$olocdec$	::=	`-obs_locations(`ι_0`,`$atlist_1$`,`$atlist_2$`).`	
$ousedec$	::=	`-obs_usage(`ι_0, ous`).`	
$opredec$	::=	`-obs_predicates([`f_1^{op}`/`i_1`,` $\ldots$ `,`f_n^{op}`/`i_n`]).`	$n \geq 0$
$obsdec$	::=	$ousedec \mid opredec$	
$obsdecs$	::=	$olocdec\ obsdec^*$	
$obsspec$	::=	$obsdecs\ opdef^*\ obsedge^+$	

Table 3.1. Erlang/Obs *syntax.*

```
-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}`(`$obsge_1$`,...,`$obsge_n$`)`, are inspired by Prolog and optionally have one or more arguments. As a side effect, an observer predicate binds unbound observer variables, that occurr 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([f_1^op/integer_1,...,f_n^op/integer_n]).
```

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

$$\begin{aligned}&\iota\texttt{(}u_1\texttt{,}\ldots\texttt{,}u_m\texttt{) when } h_1 \texttt{ -> } oce_1\texttt{;}\\ &\qquad\vdots\\ &\iota\texttt{(}u_1\texttt{,}\ldots\texttt{,}u_m\texttt{) when } h_n \texttt{ -> } oce_n\texttt{.}\end{aligned}$$

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

- $u_1, \ldots, u_m$ 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

  $$\texttt{\{next\_state,}\iota'\texttt{(}d_1\texttt{,}\ldots\texttt{,}d_k\texttt{)\}}$$

  which represents a transition to the observer state $\iota'\texttt{(}d_1\texttt{,}\ldots\texttt{,}d_k\texttt{)}$ (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 ERLANG/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

$$\texttt{-obs\_usage(}\iota_0\texttt{,}ous\texttt{)}$$

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 all runs satisfy 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 predicates 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.

Match variables for computation steps: $\langle l, \sigma \rangle \xrightarrow{a(\overline{d})/\overline{b}} \langle l', \sigma' \rangle$

`Eventtype`	is the event type a of the occurring input event,
`Eventpars`	is the sequence $\overline{d}$ of parameters of the occurring input event,
`Outevents`	is the sequence of occuring output events $\overline{b}$,
`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`,...,`u_m`) when` g `->` $\overline{v = \mathfrak{e}}$, $\overline{\mathfrak{b}}$, `{next_state,`l'`}`

`Edgevars`	is the local variables $u_1, \ldots, u_n$ of the Erlang/Efsm edge clause,
`Guard`	is the guard expression g of the Erlang/Efsm edge clause,
`Assigns`	is the sequence of bindings $\overline{v = \mathfrak{e}}$ of the Erlang/Efsm edge clause,
`Outexprs`	is the sequence of output expressions $\overline{\mathfrak{b}}$ 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 | u	
ope	::=	bv | u | mv | `[`ope_1`|`ope_2`]` | `{`ope_1`,...,`ope_n`}`	$n > 0$
	|	cp`=`ope	
	|	$guardop$`(`$ope_1, \ldots, ope_n$`)`	$n > 0$
	|	mf`(`$ope_1, \ldots, ope_n$`)`	$n \geq 0$
$opdef$	::=	f^{op}`(`u_1`,...,`u_n`) ->` ope`.`	$n \geq 0$

Table 3.2. ERLANG/OBS *observer predicate definition syntax.*

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 $u_1, \ldots, u_n$ 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

$$f^{op}\texttt{(}u_1\texttt{,}\ldots\texttt{,}u_n\texttt{) ->}\ ope\texttt{.}$$

where f^{op} is the name of the observer predicate, $u_1, \ldots, u_n$ 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 ($guardop$`(`$ope_1, \ldots, ope_n$`)`) , and match functions (mf`(`$ope_1, \ldots, ope_n$`)`).

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 occurence of a computation step. `is_sourceval/2` can be defined as

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

`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., $Fun(x, a_1, a_2, a_3)$ 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`($Assign, Var_1, Var_2$) returns $Assign$ if it is a binding of Var_2 to some expression that includes Var_1, otherwise the empty list is returned.

`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.

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.

□

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`.

□

For each match variable in Section 3.5.2 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

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.*

```
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 `{`$l_{i-1}, a_i, \overline{u_i}, g_i$`}`, uniquely identifying a single edge clause

$$l_{i-1}\texttt{(}a_i\texttt{,}u_{i1}\texttt{,...,}u_{im}\texttt{) when } g_i \texttt{ -> } \overline{v_i = \mathfrak{e}_i},\ \overline{\mathfrak{b}_i},\ \texttt{\{next\_state,}l_i\texttt{\}}$$

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 $u_{i1}, \ldots, u_{im}$, `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 • to represent the initial observer location and the symbol ◎ 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.

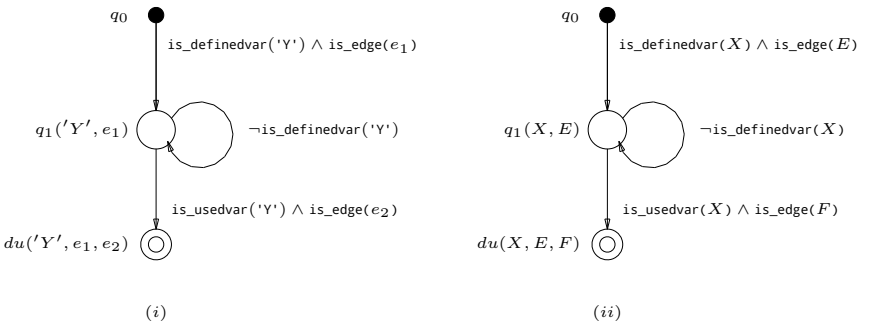


Figure 3.6. Examples of (i) observer monitoring definition (on edge e_1) and use (on edge e_2) of state variable Y, (ii) observer monitoring definition-use for *any* triples of edges and state variables.

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/Efsm 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 edges. Here e_i is a short hand notation for a tuple $\{l_{i-1}, a_i, \overline{u_i}, g_i\}$, 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.

□

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

$$\langle oe, \overline{b}, \varsigma, \rho \rangle$$

where

- oe is an Erlang/Obs expression,
- $\overline{b}$ is a sequence of output events. Since $\overline{b}$ is never used, we let it always be the empty sequence,
- $\varsigma \in MV \rightarrow D$ 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,

□

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 all 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/Efsm 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 requires certain attention and a number of new transition rules since they do not occur in Erlang/Efsm. The grammar is

found in Table 3.2. An observer predicate is defined by a clause

$$f^{op}\texttt{(}u_1\texttt{, ... ,}u_n\texttt{) -> }ope\texttt{.}$$

where f^{op} is the name of the observer predicate, u_1`, ... ,`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*,
- $guardop(ope_1, \ldots, ope_n)$, i.e., application of a guard operator over expressions $ope_1, \ldots, ope_n$, see Erlang/Efsm syntax, Table 2.1, and
- $mf(ope_1, \ldots, ope_n)$, i.e., application of a match function over expressions $ope_1, \ldots, ope_n$, see Section 3.5.2.

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

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

where the resulting local environment ρ' includes bindings to all unbound observer variables in $obsge_1$`, ... ,`$obsge_n$. How observer predicates bind unbound observer variables to Erlang/Efsm 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

$$\frac{\langle \texttt{to\_bool(L=TargetLoc)}, \varsigma, \rho\rangle \Rightarrow \langle bool, \epsilon, \varsigma, \rho'\rangle}{\langle \texttt{is\_targetloc(L)}, \varsigma, \rho\rangle \Rightarrow \langle bool, \epsilon, \varsigma, \rho'\rangle}$$

where the pattern match `L=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

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

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}.
```

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.

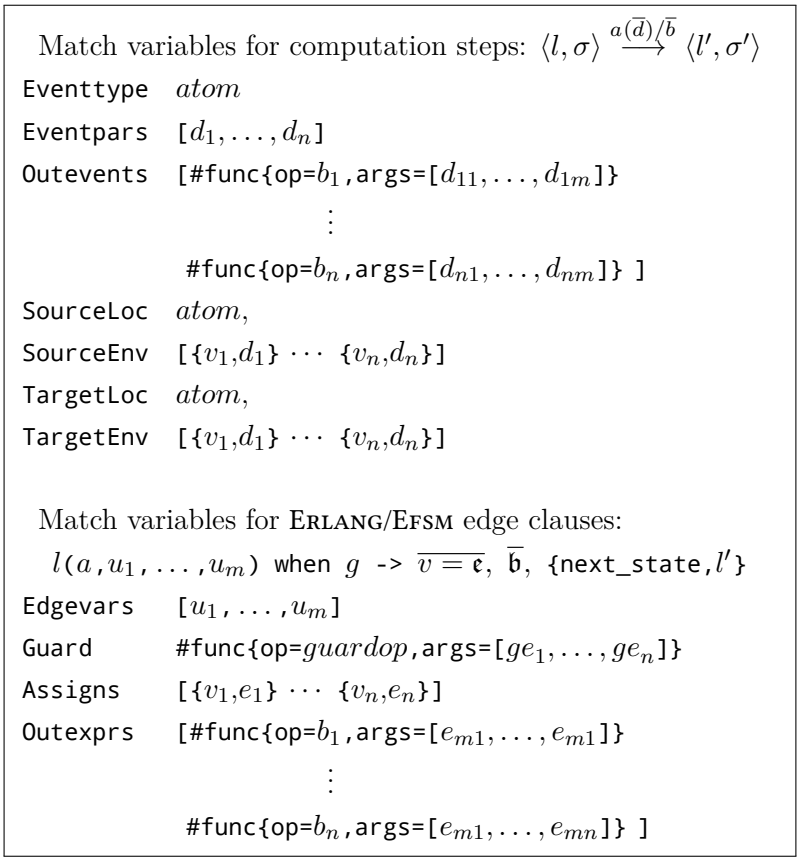


Figure 3.7. Representation of the match variables in Figure 3.4 as Erlang/Efsm expressions.

`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 `[`$ge_1, \ldots, ge_n$`]` 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/Efsm 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/Efsm 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 folllows.

$$\frac{\langle e_1, \varsigma, \rho\rangle \Rightarrow \langle d, \epsilon, \varsigma, \rho\rangle \quad \langle e_2, \varsigma, \rho\rangle \Rightarrow \langle \texttt{[}d_1, \ldots, d_n\texttt{]}, \epsilon, \varsigma, \rho\rangle \quad \exists i : 1 \leq i \leq n :: match\,(d, d_i, \rho, \rho)}{\langle \texttt{member(}e_1, e_2\texttt{)}, \varsigma, \rho\rangle \Rightarrow \langle \texttt{true}, \epsilon, \varsigma, \rho\rangle}$$

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,

$$\frac{\langle e_1, \varsigma, \rho\rangle \Rightarrow \langle d, \epsilon, \varsigma, \rho\rangle \quad \langle e_2, \varsigma, \rho\rangle \Rightarrow \langle \texttt{[}d_1, \ldots, d_n\texttt{]}, \epsilon, \varsigma, \rho\rangle \quad \forall i, 1 \leq i \leq n : \neg match\,(d, d_i, \rho, \rho)}{\langle \texttt{member(}e_1, e_2\texttt{)}, \varsigma, \rho\rangle \Rightarrow \langle \texttt{false}, \epsilon, \varsigma, \rho\rangle}$$

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.

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

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

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

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 3.6 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 in Section 3.5.2. □

`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.

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

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

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

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

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

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/EFSM 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/EFSM 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/EFSM 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 $\iota(d_1, \ldots, d_m)$ are defined by observer edge clauses

$$\begin{aligned} &\iota(u_1, \ldots, u_m) \texttt{ when } h_1 \texttt{ -> } oce_1\texttt{;} \\ &\qquad\vdots \\ &\iota(u_1, \ldots, u_m) \texttt{ when } h_n \texttt{ -> } oce_n\texttt{.} \end{aligned}$$

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

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

where oe' must always be a tuple of form `{next_state,`ι'`(`$d'_1, \ldots, d'_n$`)}`. Similarly as for configuration access functions in Erlang/Efsm, only the arguments are evaluated in an observer state, ι'`(`$d'_1, \ldots, d'_n$`)`.

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 3.2 An Erlang/Obs specification for an Erlang/Efsm specification defines an observer, which is a tuple $\langle \Sigma, \mathcal{Q}, q_0, \mathcal{Q}_f, \rightarrow \rangle$, where

- Σ is the set of annotated computation steps of the Erlang/Efsm specification,
- $\mathcal{Q}$ is the set of all terms of form $\iota\texttt{(}d_1\texttt{,}\ldots\texttt{,}d_n\texttt{)}$, where ι is an observer location declared in the Erlang/Obs specification, and $d_1\texttt{,}\ldots\texttt{,}d_n$ are Erlaqng values,
- q_0 is the term $\iota_0\texttt{()}$, where ι_0 is the declared initial observer location,
- $\mathcal{Q}_f$ is the set of all terms of form $\iota_f\texttt{(}d_1\texttt{,}\ldots\texttt{,}d_n\texttt{)}$, where ι_f is a declared observer stop location, and
- $\rightarrow\subseteq \mathcal{Q}\times\Sigma\mathcal{Q}$ is the observer transition relation. It consists of triples of form $\langle \iota\texttt{(}d_1\texttt{,}\ldots\texttt{,}d_m\texttt{)}, \varsigma, \iota'\texttt{(}d'_1\texttt{,}\ldots\texttt{,}d'_n\texttt{)}\rangle$, which we will write as

  $$\iota\texttt{(}d_1\texttt{,}\ldots\texttt{,}d_m\texttt{)} \xrightarrow{\varsigma} \iota'\texttt{(}d'_1\texttt{,}\ldots\texttt{,}d'_n\texttt{)},$$

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

  $$\langle \iota\texttt{(}d_1\texttt{,}\ldots\texttt{,}d_m\texttt{)}, \varsigma, \bullet\rangle \Rightarrow \langle \texttt{\{next\_state,}\iota'\texttt{(}d'_1\texttt{,}\ldots\texttt{,}d'_n\texttt{)\}}, \epsilon, \varsigma, \rho'\rangle \ .$$

□

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

Definition 3.3 Let $\langle l_0, \bullet\rangle \xrightarrow{a_1(\overline{d_1})/\overline{b_1}} \cdots \xrightarrow{a_n(\overline{d_n})/\overline{b_n}} \langle l_n, \sigma_n\rangle$ be a run of an EFSM, and for $i = 1,\ldots,n$, let ς_i represent the annotated computation step derived from the ith computation step, of form

$$\left\langle \begin{array}{l} \langle l_{i-1}, \sigma_{i-1}\rangle \xrightarrow{a_i(\overline{d_i})/\overline{b_i}} \langle l_i, \sigma_i\rangle, \\ l_{i-1}\texttt{(}a\texttt{,}u_1\texttt{,}\ldots\texttt{,}u_m\texttt{)}\ \texttt{when}\ g\ \texttt{->}\ \overline{v = \mathfrak{e}},\ \overline{\mathfrak{b}},\ \texttt{\{next\_state,}l_i\texttt{\}} \end{array} \right\rangle$$

We say that the run *covers* a coverage item $q_f \in \mathcal{Q}_f$ if there is a run $q_0 \xrightarrow{\varsigma_1} \cdots \xrightarrow{\varsigma_n} q_n$ 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. □

3.7 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 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.

Definition 3.4 A *symbolic structural (observer) state* is a tuple

$$\langle oe, H, \overline{b}, \varsigma^s, \rho \rangle$$

where

- oe is an ERLANG/OBS expression,
- H is a *superposition condition*, i.e., a condition imposed by the observer onto an EFSM under which this symbolic structural state can be reached. A superposition condition is analogous to the path condition imposed by the symbolic structural states of ERLANG/EFSM in a symbolic run.
- $\overline{b}$ is a sequence of output expressions. Since $\overline{b}$ is never used, we let it always be the empty sequence,
- $\varsigma^s \in V \to \mathfrak{E}$ is a global symbolic environment with bindings to the match variables V, and
- $\rho \in U \to \mathfrak{E}$ is a local symbolic environment with bindings to the observer variables U.

□

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

Definition 3.5 A *symbolic annotated computation step* is a pair

$$\left\langle \begin{array}{l} \langle l, \sigma, G \rangle \stackrel{a(\overline{p})/\overline{\mathfrak{b}}}{\longrightarrow} \langle l', \sigma', G' \rangle, \\ l\texttt{(}a\texttt{,}u_1\texttt{,}\ldots\texttt{,}u_m\texttt{) when } g \texttt{ -> } \overline{v = \mathfrak{e}}\texttt{, } \overline{\mathfrak{b}}\texttt{, \{next\_state,}l'\texttt{\}} \end{array} \right\rangle$$

that consists of

- the symbolic computation step $\langle l, \sigma, G \rangle \stackrel{a(\overline{p})/\overline{\mathfrak{b}}}{\longrightarrow} \langle l', \sigma', G' \rangle$ generated in response to $a(\overline{p})$, and
- the (syntactic) ERLANG/EFSM edge clause

  $$l\texttt{(}a\texttt{,}u_1\texttt{,}\ldots\texttt{,}u_m\texttt{) when } g \texttt{ -> } \overline{v = \mathfrak{e}}\texttt{, } \overline{\mathfrak{b}}\texttt{, \{next\_state,}l'\texttt{\}}$$

  from which the symbolic computation step is derived.

□

A symbolic transition between two symbolic structural states

$$\langle oe, \varsigma^s, \rho \rangle \Rightarrow \left\langle \mathfrak{e}, H, \epsilon, \varsigma^s, \rho' \right\rangle$$

denotes that *oe* can be evaluated in a global environment ς^s and local environment ρ to an ERLANG/EFSM normal form expression $\mathfrak{e}$, 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 3.7 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,SourcEnv) 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

$$\iota_0\texttt{() when equal\_vars(V1,V2) ->}\ oce\texttt{.}$$

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

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

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

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

also results in an expression $\mathfrak{e}_1==\mathfrak{e}_2$ where both $\mathfrak{e}_1$ and $\mathfrak{e}_2$ are expressions with symbolic parameters. Then, given that `V1=/=V2` is satisfied, we can conclude that the necessary condition for the observer predicate to evaluate to `true` is that (the superposistion condition) $\mathfrak{e}_1==\mathfrak{e}_2$ is satisfied. □

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

$$f^{op}\texttt{(}u_1\texttt{,} \ldots \texttt{,}u_n\texttt{) -> }ope\texttt{.}$$

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

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

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

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

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

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.

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/EFSM 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.

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

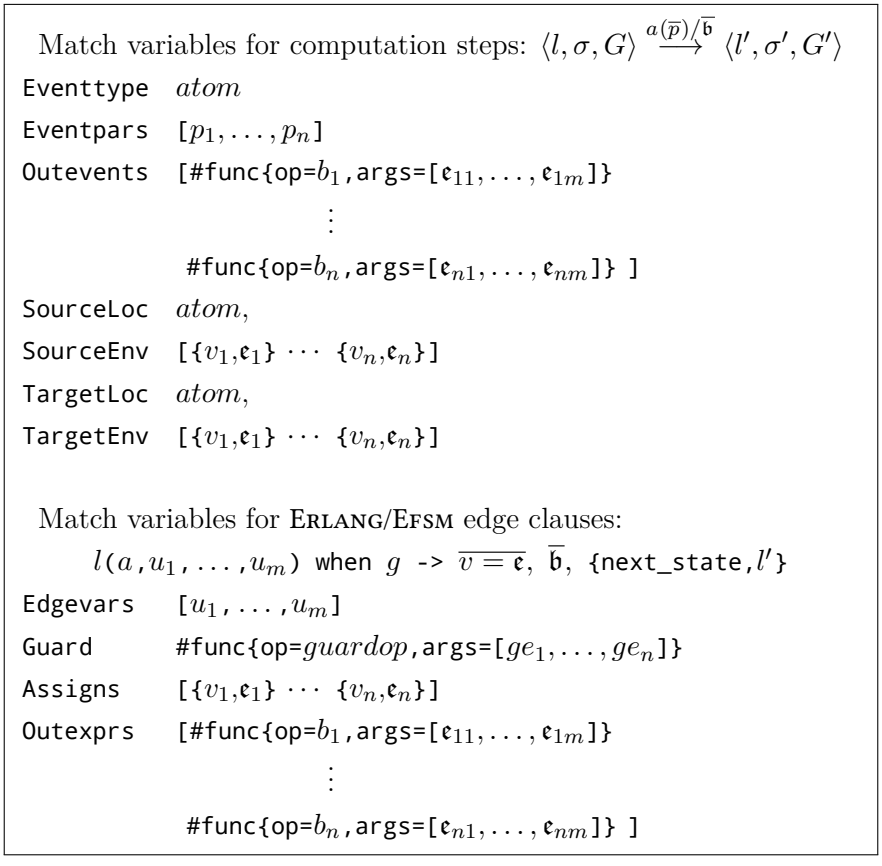


Figure 3.8. Representation of the match variables in Figure 3.4 as Erlang/Efsm expressions with symbolic execution.

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,

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

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,

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

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

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

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.

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

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

$$\frac{\begin{array}{c}\langle e_1, \varsigma^s, \rho\rangle \Rightarrow \langle \mathfrak{e}_1, H', \epsilon, \varsigma^s, \rho\rangle \\ \langle e_2, \varsigma^s, \rho\rangle \Rightarrow \langle \texttt{[\{}\mathfrak{e}_{11}, \mathfrak{e}_{12}\texttt{\},}\ldots\texttt{,\{}\mathfrak{e}_{n1}, \mathfrak{e}_{n2}\texttt{\}]}, H'', \epsilon, \varsigma^s, \rho\rangle \\ \forall j, 1 \leq j \leq n : \neg match\,(\mathfrak{e}_1, \mathfrak{e}_{j1}, \rho, H_{ji}, \rho)\end{array}}{\langle \texttt{lookup(}e_1, e_2\texttt{)}, \varsigma^s, \rho\rangle \Rightarrow \langle \texttt{false}, H' \wedge H'' \wedge H_{11} \wedge \ldots \wedge H_{n1}, \epsilon, \varsigma^s, \rho\rangle}$$

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

$$\frac{\begin{array}{c}\langle e_1, \varsigma^s, \rho\rangle \Rightarrow \langle u, H', \epsilon, \varsigma^s, \rho\rangle \\ u \text{ is an observer variable, not defined in } \rho \\ \langle e_2, \varsigma^s, \rho\rangle \Rightarrow \langle \texttt{[\{}\mathfrak{e}_{11}, \mathfrak{e}_{12}\texttt{\},}\ldots\texttt{,\{}\mathfrak{e}_{n1}, \mathfrak{e}_{n2}\texttt{\}]}, \texttt{true}, \epsilon, \varsigma^s, \rho\rangle\end{array}}{\langle \texttt{lookup(}e_1, e_2\texttt{)}, \varsigma^s, \rho\rangle \Rightarrow \langle \mathfrak{e}_{i2}, \texttt{true}, \epsilon, \varsigma^s, \rho[u \mapsto \mathfrak{e}_{i2}]\rangle}$$

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* 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 $\iota\texttt{(}\mathfrak{e}_1\texttt{,}\ldots\texttt{,}\mathfrak{e}_m\texttt{)}$ are defined by observer edge clauses

$$\begin{array}{c}\iota\texttt{(}u_1\texttt{,}\ldots\texttt{,}u_m\texttt{)}\ \texttt{when}\ h_1\ \texttt{->}\ oce_1\texttt{;} \\ \vdots \\ \iota\texttt{(}u_1\texttt{,}\ldots\texttt{,}u_m\texttt{)}\ \texttt{when}\ h_n\ \texttt{->}\ oce_n\texttt{.}\end{array}$$

Each of these clauses are evaluated by first matching the expressions $\mathfrak{e}_1, \ldots, \mathfrak{e}_m$ with the observer variables $u_1\texttt{,}\ldots\texttt{,}u_m$, and if the match succeeds the observer guard expression h_k is evaluated. For *some* clause k with all observer variables, $u_1\texttt{,}\ldots\texttt{,}u_m$ matching and observer guard expression h_k evaluating to `true`, the oce_k expression is evaluated. The symbolic transition rule is

$$\frac{\begin{array}{c}match\,(u_{k1}\texttt{,}\ldots\texttt{,}u_{km}, \mathfrak{e}_1, \ldots, \mathfrak{e}_m, \bullet, \rho') \\ \langle h_k, \varsigma^s, \rho\rangle \Rightarrow \langle \texttt{true}, H, \epsilon, \varsigma^s, \rho'\rangle \\ \langle oce_k, \varsigma^s, \rho''\rangle \Rightarrow \langle \mathfrak{e}', \texttt{true}, \epsilon, \varsigma^s, \rho'''\rangle\end{array}}{\langle \iota(\mathfrak{e}_1\texttt{,}\ldots\texttt{,}\mathfrak{e}_m), \varsigma^s, \rho_0\rangle \Rightarrow \langle \mathfrak{e}', H, \epsilon, \varsigma^s, \rho_m\rangle}$$

where $\mathfrak{e}'$ always must be a tuple of form $\texttt{\{next\_state,}\iota'\texttt{(}\mathfrak{e}'_1\texttt{,}\ldots\texttt{,}\mathfrak{e}'_n\texttt{)\}}$. Similarly as for configuration access functions in ERLANG/EFSM, only the arguments are evaluated in the observer state, $\iota'\texttt{(}\mathfrak{e}'_1\texttt{,}\ldots\texttt{,}\mathfrak{e}'_n\texttt{)}$.

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/EFSM definition, with symbolic semantics as in Section 2.9.

Let a *symbolic observer state* $\mathfrak{q}$ be a term of form $\iota\texttt{(}\mathfrak{e}_1\texttt{,}\ldots\texttt{,}\mathfrak{e}_n\texttt{)}$, where ι is an observer location declared in the ERLANG/OBS specification and $\mathfrak{e}_1\texttt{,}\ldots\texttt{,}\mathfrak{e}_n$ are ERLANG/EFSM normal form expressions. Let the *initial symbolic observer state* be the term $\langle\iota_0\texttt{()},\texttt{true}\rangle$, where ι_0 is the declared initial observer location. Let an *accepting symbolic observer state* $\mathfrak{q}_f$ be a term of form $\iota_f\texttt{(}\mathfrak{e}_1\texttt{,}\ldots\texttt{,}\mathfrak{e}_n\texttt{)}$, where ι_f is a declared observer stop location. Let a *superposition condition* H be an expression with symbolic parameters from the EFSM. Let a *conditioned observer state* be a term of form $\langle\mathfrak{q},H\rangle$ $\mathfrak{q}$ is a symbolic observer state and H is a superposition condition. Let an *accepting conditioned observer state* be a conditioned observer state $\langle\mathfrak{q}_f,H\rangle$, where $\mathfrak{q}_f \in \mathcal{Q}_f$. We will sometimes use the term *conditioned coverage item* for accepting conditioned observer state.

Let a *conditioned observer step* be a triple of form

$$\langle\iota\texttt{(}\mathfrak{e}_1\texttt{,}\ldots\texttt{,}\mathfrak{e}_m\texttt{)},H\rangle \quad \overset{\varsigma^s}{\hookrightarrow} \quad \langle\iota'\texttt{(}\mathfrak{e}'_1\texttt{,}\ldots\texttt{,}\mathfrak{e}'_n\texttt{)},H\wedge H'\rangle \quad ,$$

where ς^s hold bindings of match variables to ERLANG/EFSM normal form expressions that representing a current symbolic annotated computation step, such that the rules in the symbolic semantics for observers allow to derive the symbolic transition

$$\langle\iota\texttt{(}\mathfrak{e}_1\texttt{,}\ldots\texttt{,}\mathfrak{e}_m\texttt{)},\varsigma^s,\bullet\rangle \Rightarrow \langle\texttt{\{next\_state,}\iota'\texttt{(}\mathfrak{e}'_1\texttt{,}\ldots\texttt{,}\mathfrak{e}'_n\texttt{)\}},H',\epsilon,\varsigma^s,\rho\rangle \quad .$$

Intuitively, whenever the observer is in a conditioned observer state $\langle\mathfrak{q},H\rangle$, and the EFSM performs a symbolic computation step at the occurrence of an input event, the corresponding symbolic annotated computation step ς^s is generated as input to the observer, which makes a conditioned observer step of form $\langle\mathfrak{q},H\rangle \overset{\varsigma^s}{\hookrightarrow} \langle\mathfrak{q}',H\wedge H'\rangle$ thereby moving to the conditioned observer state $\langle\mathfrak{q}',H\wedge H'\rangle$.

We can also provide a definition of coverage for the symbolic case.

Definition 3.6 Let $\langle l_0,\bullet,\texttt{true}\rangle \overset{a_1(\overline{p_1})/\overline{\mathfrak{b}_1}}{\longrightarrow} \cdots \overset{a_n(\overline{p_n})/\overline{\mathfrak{b}_n}}{\longrightarrow} \langle l_n,\sigma_n,G_n\rangle$ be a symbolic run of an EFSM, and for $i=1,\ldots,n$, let ς^s_i represent the ith

symbolic annotated computation step

$$\left\langle \begin{array}{l} \langle l_{i-1}, \sigma_{i-1}, G_{i-1}\rangle \overset{a_i(\overline{p_i})/\overline{\mathfrak{b}_i}}{\longrightarrow} \langle l_i, \sigma_i, G_i\rangle, \\ l_{i-1}\texttt{(}a\texttt{,}u_1\texttt{,}\ldots\texttt{,}u_m\texttt{) when } g \texttt{ -> } \overline{v = \mathfrak{e}}\texttt{, } \overline{\mathfrak{b}}\texttt{, \{next\_state,}l_i\texttt{\}} \end{array} \right\rangle$$

Let $\mathfrak{q}_f$ be an accepting symbolic observer state. Then we say that this symbolic run *symbolically covers* $\langle \mathfrak{q}_f, H_n\rangle$ if there is a sequence of conditioned observer steps

$$\langle \mathfrak{q}_0, \texttt{true}\rangle \overset{\varsigma_1^s}{\hookrightarrow} \langle \mathfrak{q}_1, H_1\rangle \overset{\varsigma_2^s}{\hookrightarrow} \langle \mathfrak{q}_2, H_2\rangle \quad \cdots \quad \langle \mathfrak{q}_{n-1}, H_{n-1}\rangle \overset{\varsigma_n^s}{\hookrightarrow} \langle \mathfrak{q}_n, H_n\rangle$$

such that $\mathfrak{q}_n = \mathfrak{q}_f$. We say that a symbolic test case $\langle w, G\rangle$ *symbolically covers* $\langle \mathfrak{q}_f, H_n\rangle$ if there is a symbolic run of the EFSM $\langle l_0, \bullet, \texttt{true}\rangle \overset{a_1(\overline{p_1})/\overline{\mathfrak{b}_1}}{\longrightarrow} \cdots \overset{a_n(\overline{p_n})/\overline{\mathfrak{b}_n}}{\longrightarrow} \langle l_n, \sigma_n, G_n\rangle$ over w with $G_n = G$ which symbolically covers $\langle \mathfrak{q}_f, H_n\rangle$. □

Intuitively, this definition says that all coverage items represented by an instantiation of $\mathfrak{q}_f$ that satisfies $G_n \wedge H_n$, are covered by runs of the EFSM that instantiate the symbolic run $\langle l_0, \bullet, \texttt{true}\rangle \overset{a_1(\overline{p_1})/\overline{\mathfrak{b}_1}}{\longrightarrow} \cdots \overset{a_n(\overline{p_n})/\overline{\mathfrak{b}_n}}{\longrightarrow} \langle l_n, \sigma_n, G_n\rangle$ in a way that satisfies $G_n \wedge H_n$.

A correspondence between symbolic coverage and ordinary coverage is provided by the following theorem.

Theorem 3.1 Assume an EFSM and an observer monitoring the EFSM. Then there is a run of the EFSM which covers a coverage item q_f if and only if there is a symbolic observer state $\mathfrak{q}_f$, a path condition G_n, and a superposition condition H_n such that there is a symbolic run of the EFSM, with final path condition G_n, which symbolically covers $\langle \mathfrak{q}_f, H_n\rangle$, for which there is a symbolic environment Γ such that $q_f = \Gamma(\mathfrak{q}_f)$.

□

A proof can be given by induction on computation steps. Assume that there is a run of an EFSM, of form

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

that covers a coverage item q_f with a sequence of observer steps

$$q_0 \overset{\varsigma_1}{\rightarrow} q_1 \overset{\varsigma_2}{\rightarrow} q_2 \quad \cdots \quad q_{n-1} \overset{\varsigma_n}{\rightarrow} q_n \quad .$$

and symbolic run of the same EFSM, and same sequence of transitions

$$\langle l_0, \bullet, \texttt{true}\rangle \overset{a_1(\overline{p_1})/\overline{\mathfrak{b}_1}}{\longrightarrow} \langle l_1, \sigma_1, G_1\rangle \overset{a_2(\overline{p_2})/\overline{\mathfrak{b}_2}}{\longrightarrow} \cdots \overset{a_n(\overline{p_n})/\overline{\mathfrak{b}_n}}{\longrightarrow} \langle l_n, \sigma_n, G_n\rangle$$

that symbolically covers $\langle symbobsstateF, G_n \wedge H_n \rangle$ with a sequence of conditioned observer steps

$$\langle \mathfrak{q}_0, \mathtt{true} \rangle \overset{\varsigma_1^s}{\hookrightarrow} \langle \mathfrak{q}_1, H_1 \rangle \overset{\varsigma_2^s}{\hookrightarrow} \langle \mathfrak{q}_2, H_2 \rangle \quad \cdots \quad \langle \mathfrak{q}_{n-1}, H_{n-1} \rangle \overset{\varsigma_n^s}{\hookrightarrow} \langle \mathfrak{q}_n, H_n \rangle$$

where $\mathfrak{q}_n$ is an accepting symbolic observer state $\mathfrak{q}_f$. Initially, $\langle l_0, \bullet \rangle$ and q_0 is equal to $\langle l_0, \bullet, \mathtt{true} \rangle$ and $\langle \mathfrak{q}_0, \mathtt{true} \rangle$. Let

$$a_1(\overline{d_1})/\overline{b_1} \quad a_2(\overline{d_2})/\overline{b_2} \quad \cdots \quad a_i(\overline{d_i})/\overline{b_i}$$

be the trace after i observer steps and $\overline{p_1}, \ldots, \overline{p_i}$ all input expression parameters after i conditioned observer steps. Then, $\langle l_i, \sigma_i \rangle$ and q_i is equal to $\langle l_i, \sigma_i, G_i \rangle$ and $\langle \mathfrak{q}_i, H_i \rangle$ iff $G_i \wedge H_i$ evaluates to `true` and $\mathfrak{q}_i$ evaluates to q_i for some assignment $\overline{p_1} \mapsto \overline{d_1}, \ldots, \overline{p_i} \mapsto \overline{d_i}$ of the symbolic input expression parameters and symbolic configuration parameters of form $f(\mathfrak{e}_1, \ldots \mathfrak{e}_n)$, depending on those symbolic input expression parameters.

Example 3.8 Assume an edge clause

$$l(a) \texttt{ when } (\texttt{X==}0) \texttt{ or } (\texttt{Y==}2) \texttt{ -> } \texttt{\{next\_state,}l'\texttt{\}}$$

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:

$$\begin{aligned} h_1 &= \texttt{is\_cc('X==0',false)} \\ h_2 &= \texttt{is\_cc('X==0',true)} \\ h_3 &= \texttt{is\_cc('Y==2',false)} \\ h_4 &= \texttt{is\_cc('Y==2',true)} \end{aligned}$$

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.

□

4. Coverage criteria

Observer automata, introduced in the previous chapter, control the selection of test cases. In this chapter we give examples on 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 [Broy 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 litterature, a *decision* is often refered 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)

$$l(a)\texttt{ when (X==0) or (Y==2) -> \{next\_state,}l'\texttt{\}}$$

where the guard, (X==0) 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
0	2
–	–
–	–
1	*3*

(i)

X	Y
–	–
0	3
1	2
–	–

(ii)

X	Y
0	2
0	3
1	2
1	*3*

(iii)

X	Y
–	–
0	3
1	2
1	*3*

(iv)

X	Y
–	–
0	3
1	2
1	*3*

(v)

Figure 4.1. Sufficient tests for a decision (X==0) 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) 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) 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) 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/Efsm. 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/Efsm.

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 with conditions $[r_1, \ldots, r_n]$ 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).

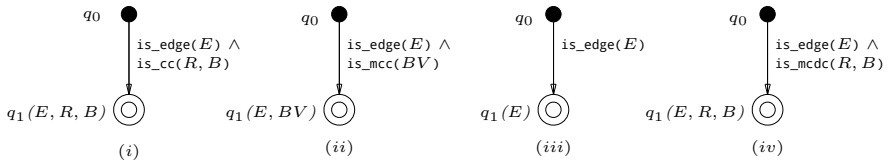


Figure 4.2. Observers for the control flow criteria on guards: (i) Condition (or Condition/Decision) Coverage, (ii) Multiple Condition Coverage, (iii) Decision Coverage (and All-Edges) and (iv) Modified Condition/Decision Coverage.

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 $e_1, \ldots, e_{10}$ 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.

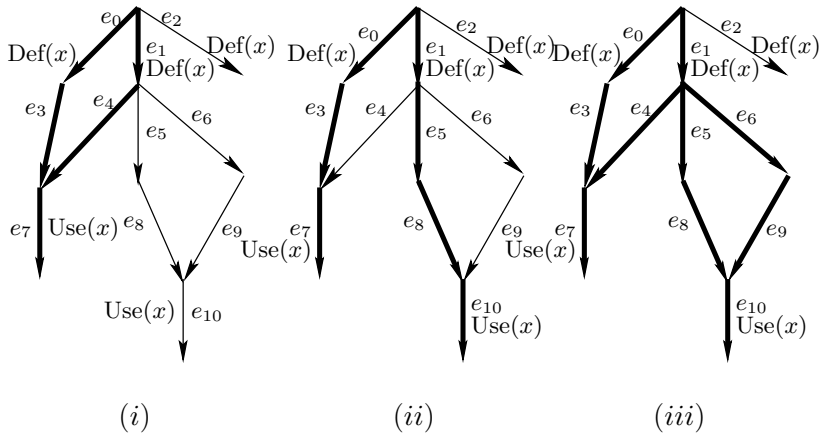


Figure 4.3. EFSM graphs, with annotations for definition and usage of a state variable x, illustrating (i) Reachable definitions coverage, (ii) Reachable uses coverage, and (iii) Definition-Use pair coverage

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.

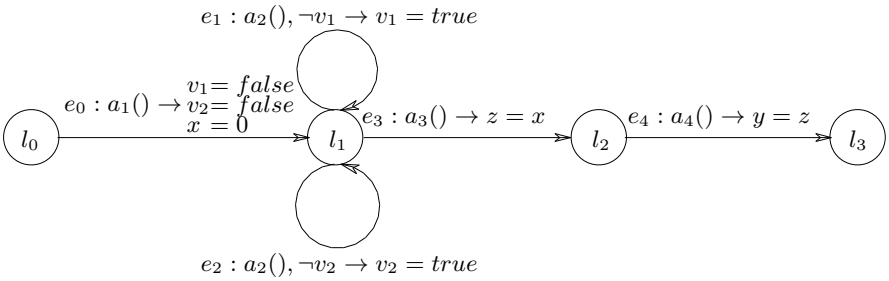


Figure 4.4. EFSM graph example for illustrating Definition context coverage and Effect pairs coverage.

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`, `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 `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).

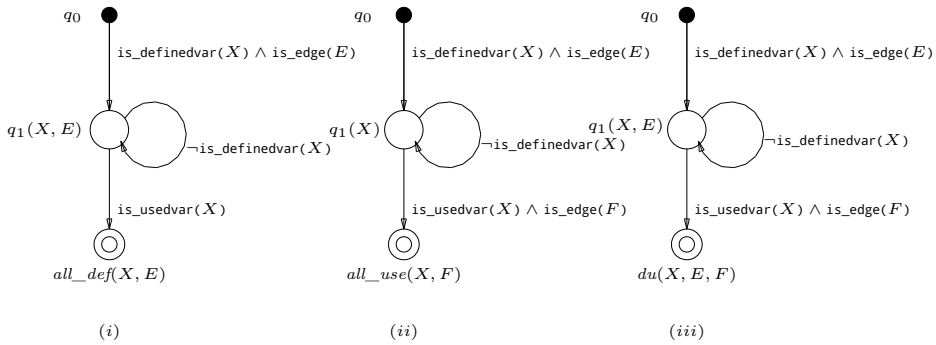


Figure 4.5. Observers for data flow criteria: (i) Reachable definitions coverage, (ii) Reachable uses coverage, and (iii) Definition-Use pair coverage.

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 99b, 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.

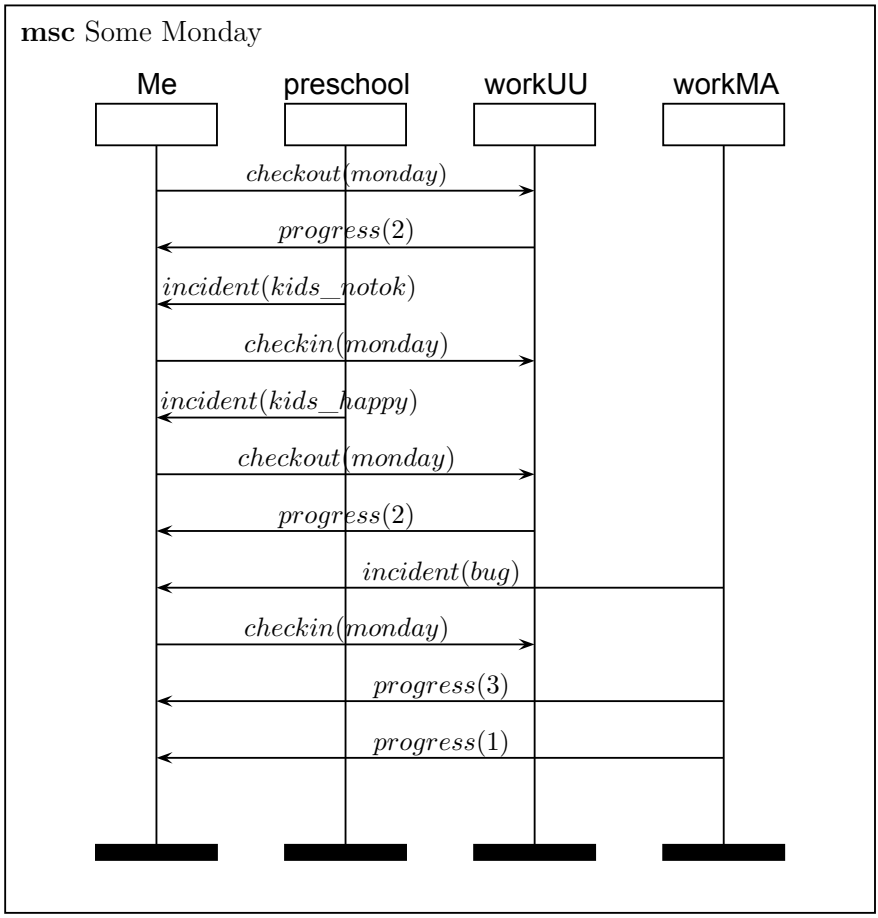


Figure 4.8. A possible sequence of events on the EFSM from Example 1.1.

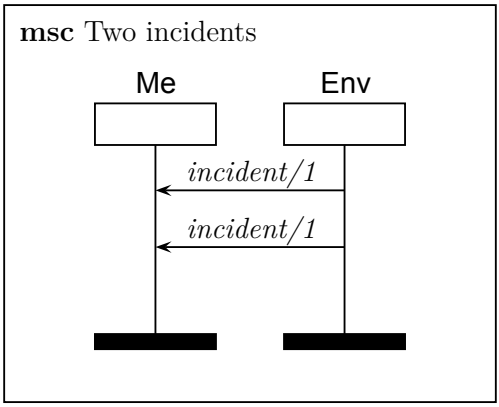


Figure 4.9. A possible MSC that can be used to generate a test suite from.

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.

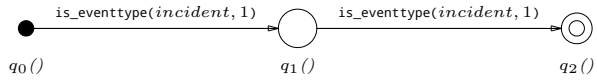


Figure 4.10. An example of an observer for selecting test cases that contains two consecutive incident events.

- 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 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 2.3, 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 representation of test cases, Section 5.2 discuss test generation algorithms, Section 5.3 suggest an efficient representation of observer states, Section 5.4 discusses the necessary mapping between the abstract model and the concrete world, and Section 5.5 outlines some further usage 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 3.1 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 a conditioned coverage item $\langle \mathfrak{q}_f, H_n \rangle$ 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 $\langle \mathfrak{q}_f, H_n \rangle$, we need to generate symbolic runs of both the EFSM and the observer, which match. The symbolic run of the EFSM $\langle l_0, \bullet, \mathtt{true} \rangle \xrightarrow{a_1(\overline{p_1})/\overline{\mathfrak{b}_1}} \cdots \xrightarrow{a_n(\overline{p_n})/\overline{\mathfrak{b}_n}} \langle l_n, \sigma_n, G_n \rangle$ should be such that there is symbolic run of the observer $\langle \mathfrak{q}_0, \mathtt{true} \rangle \overset{s_1}{\hookrightarrow} \cdots \overset{s_n}{\hookrightarrow} \langle \mathfrak{q}_n, H_n \rangle$ over the corresponding sequence of symbolic annotated computation steps. The close correspondence between these two runs suggest that the test suite generation should generate them together. We then introduce a new combined state, which we view as a *superposition* of a (symbolic) observer state onto a (symbolic) EFSM state. Moreover, since each prefix of an EFSM run can in general correspond to a set of observer runs, we will actually superpose *a set of* (symbolic) observer states onto a (symbolic) EFSM state. 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.

Example 5.1 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

$$\langle \mathtt{morning}, [\,], \mathtt{true} \rangle \xrightarrow{\mathtt{wakeup}(T_{Day})/\mathtt{checkout}(T_{Day})}$$

$$\langle \mathtt{workUU}, \begin{bmatrix} \mathtt{Progress{=}0} \\ \mathtt{Stamina{=}2} \\ \mathtt{Day{=}TDay} \end{bmatrix}, \mathtt{daytype}(T_{Day})\mathtt{==collect} \rangle \xrightarrow{\mathtt{incident}(I_1)/\mathtt{checkin}(T_{Day})}$$

$$\langle \mathtt{preschool}, \begin{bmatrix} \mathtt{Progress{=}0} \\ \mathtt{Stamina{=}1} \\ \mathtt{Day{=}TDay} \end{bmatrix}, \begin{array}{l} \mathtt{daytype}(T_{Day})\mathtt{==collect} \\ \wedge\, I_1 = \mathtt{kids\_happy} \end{array} \rangle \xrightarrow{\mathtt{incident}(I_2)/}$$

$$\langle \mathtt{end\_of\_day}, \begin{bmatrix} \mathtt{Progress{=}0} \\ \mathtt{Stamina{=}0} \\ \mathtt{Day{=}TDay} \end{bmatrix}, \begin{array}{l} \mathtt{daytype}(T_{Day})\mathtt{==collect} \\ \wedge\, I_1\mathtt{==kids\_happy} \\ \wedge\, I_2\mathtt{==kids\_notok} \end{array} \rangle$$

from which we can extract a symbolic test case as the tuple $\langle w, G \wedge H \rangle$, where

- w is the symbolic trace with input and output expressions:

$$\texttt{wakeup}(TDay)/\texttt{checkout}(TDay)$$
$$\texttt{incident}(I_1)/\texttt{checkin}(TDay)$$
$$\texttt{incident}(I_2)/,$$

- G is the path condition:

$$\texttt{daytype}(TDay)\texttt{==collect} \wedge I_1 = \texttt{kids\_happy} \wedge I_2 = \texttt{kids\_notok},$$

  and
- H is the superposition condition `true`.

This test case symbolically covers

$$\texttt{visitedloc(workUU)},$$
$$\texttt{visitedloc(preschool)} \text{ and}$$
$$\texttt{visitedloc(end\_of\_day)}$$

provided

$$\texttt{daytype}(TDay)\texttt{==collect} \wedge I_1 = \texttt{kids\_happy} \wedge I_2 = \texttt{kids\_notok} \ .$$

□

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 $\langle l, \sigma, G \rangle$. □

Definition 5.2 A *symbolic superpositioned computation step* is a triple

$$\langle \langle l, \sigma, G \rangle \parallel \mathcal{Q} \parallel \omega \rangle \overset{a(\overline{p})/\overline{\mathfrak{b}}}{\rightsquigarrow} \langle \langle l', \sigma', G' \rangle \parallel \mathcal{Q}' \parallel \omega \cdot a(\overline{p})/\overline{\mathfrak{b}} \rangle$$

where

- $\langle l, \sigma, G \rangle \overset{a(\overline{p})/\overline{\mathfrak{b}}}{\longrightarrow} \langle l', \sigma', G' \rangle$ is a symbolic computation step; we let ς^s represent the corresponding symbolic annotated computation step,
- $\mathcal{Q}' = \left\{ \langle \mathfrak{q}', H' \rangle \;\middle|\; \begin{array}{l} \langle \mathfrak{q}, H \rangle \xrightarrow{\varsigma^s} \langle \mathfrak{q}', H' \rangle \\ \langle \mathfrak{q}, H \rangle \in \mathcal{Q} \end{array} \right\}$ is the set of conditioned observer states to which the observer can make conditioned observer steps from some conditioned observer state $\langle \mathfrak{q}, H \rangle$ 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

$$\langle\langle l_0, \bullet, \mathtt{true}\rangle \parallel \{\langle \mathfrak{q}_0, \mathtt{true}\rangle\} \parallel \epsilon\rangle \overset{a_1(\overline{p_1})/\overline{\mathfrak{b}_1}}{\rightsquigarrow} \cdots \overset{a_n(\overline{p_n})/\overline{\mathfrak{b}_n}}{\rightsquigarrow} \langle\langle l_n, \sigma_n, G_n\rangle \parallel \mathcal{Q}_n \parallel \omega_n\rangle$$

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 $\mathcal{Q}$.

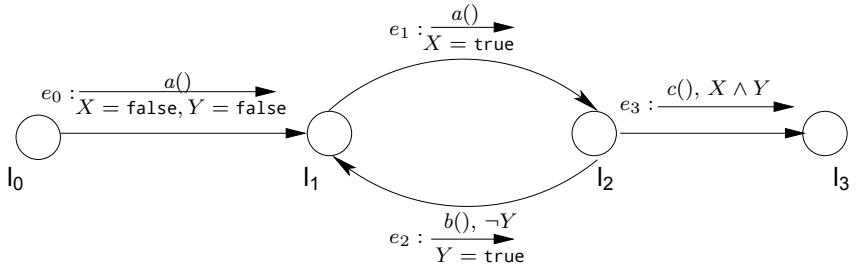


Figure 5.1. A graphical representation of a simple EFSM with a unique name (e_x) associated to each edge.

Example 5.2 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

$$\langle\langle l_0, \bullet, \mathtt{true}\rangle \parallel \{\langle q_0, \mathtt{true}\rangle\} \parallel \epsilon\rangle \overset{a()/}{\rightsquigarrow}$$

$$\langle\langle l_1, \left\{ \begin{array}{l} X{:=}\mathtt{false} \\ Y{:=}\mathtt{false} \end{array} \right\}, \mathtt{true}\rangle \parallel \left\{ \begin{array}{l} \langle q_0, \mathtt{true}\rangle \\ \langle q_1('X', e_0), \mathtt{true}\rangle \\ \langle q_1('Y', e_0), \mathtt{true}\rangle \end{array} \right\} \parallel a()/\rangle \overset{a()/}{\rightsquigarrow}$$

$$\langle\langle l_2, \left\{ \begin{array}{l} X{:=}\mathtt{true} \\ Y{:=}\mathtt{false} \end{array} \right\}, \mathtt{true}\rangle \parallel \left\{ \begin{array}{l} \langle q_0, \mathtt{true}\rangle \\ \langle q_1('X', e_1), \mathtt{true}\rangle \\ \langle q_1('Y', e_0), \mathtt{true}\rangle \end{array} \right\} \parallel a()/\cdot a()/\rangle \overset{b()/}{\rightsquigarrow}$$

$$\langle\langle l_1, \left\{ \begin{array}{l} X{:=}\mathtt{true} \\ Y{:=}\mathtt{true} \end{array} \right\}, \mathtt{true}\rangle \parallel \left\{ \begin{array}{l} \langle q_0, \mathtt{true}\rangle \\ \langle q_1('X', e_1), \mathtt{true}\rangle \\ \langle q_1('Y', e_2), \mathtt{true}\rangle \\ \langle du('Y', e_0, e_2), \mathtt{true}\rangle \end{array} \right\} \parallel a()/\cdot a()/\cdot b()/\rangle \overset{a()/}{\rightsquigarrow}$$

$$\langle\langle l_2, \left\{ \begin{array}{l} X{:=}\mathtt{true} \\ Y{:=}\mathtt{true} \end{array} \right\}, \mathtt{true}\rangle \parallel \left\{ \begin{array}{l} \langle q_0, \mathtt{true}\rangle \\ \langle q_1('X', e_1), \mathtt{true}\rangle \\ \langle q_1('Y', e_2), \mathtt{true}\rangle \\ \langle du('Y', e_0, e_2), \mathtt{true}\rangle \end{array} \right\} \parallel a()/\cdot a()/\cdot b()/\cdot a()/\rangle \overset{c()/}{\rightsquigarrow}$$

$$\langle\langle l_3, \left\{ \begin{array}{l} X{:=}\mathtt{true} \\ Y{:=}\mathtt{true} \end{array} \right\}, \mathtt{true}\rangle \parallel \left\{ \begin{array}{l} \langle q_0, \mathtt{true}\rangle \\ \langle du('Y', e_0, e_2), \mathtt{true}\rangle \\ \langle du('X', e_1, e_3), \mathtt{true}\rangle \\ \langle du('Y', e_2, e_3), \mathtt{true}\rangle \end{array} \right\} \parallel a()/\cdot a()/\cdot b()/\cdot a()/\cdot c()/\rangle$$

□

Let $\mathcal{Q}_f$ denote the set of conditioned observer states $\langle \mathfrak{q}_f, H\rangle$, where $\mathfrak{q}_f$ is an accepting symbolic observer state.

In Definition 3.6 we defined how a symbolic run of an EFSM symbolically covers a coverage item. It follows from the definition of symbolic superposition run that if

$$\langle\langle l_0, \bullet, \mathtt{true}\rangle \parallel \{\langle \mathfrak{q}_0, \mathtt{true}\rangle\} \parallel \epsilon\rangle \overset{a_1(\overline{p_1})/\overline{\mathfrak{b}_1}}{\rightsquigarrow} \cdots \overset{a_n(\overline{p_n})/\overline{\mathfrak{b}_n}}{\rightsquigarrow} \langle\langle l_n, \sigma_n, G_n\rangle \parallel \mathcal{Q} \parallel \omega_n\rangle$$

is a symbolic superpositioned run, then the symbolic run

$$\langle l_0, \bullet, \mathtt{true}\rangle \xrightarrow{a_1(\overline{p_1})/\overline{\mathfrak{b}_1}} \langle l_1, \sigma_1, G_1\rangle \xrightarrow{a_2(\overline{p_2})/\overline{\mathfrak{b}_2}} \cdots \xrightarrow{a_n(\overline{p_n})/\overline{\mathfrak{b}_n}} \langle l_n, \sigma_n, G_n\rangle$$

of the EFSM symbolically covers an accepting conditioned observer state $\langle \mathfrak{q}_n, H_n\rangle$ whenever $\langle \mathfrak{q}_n, H_n\rangle \in \mathcal{Q}_n$. By Theorem 3.1, a coverage item q_f which instantiates $\mathfrak{q}_n$ is obtained by applying a symbolic environment Γ such that $q_f = \Gamma(\mathfrak{q}_f)$.

Note that in general a single accepting symbolic observer state may cover several coverage items. Further note that a single symbolic run of the EFSM may cover several accepting symbolic observer states. For example, in Example 5.2 the two possible definition-use pairs represented by the accepting symbolic observer states $du('X', e_0, e_2)$ and $du('Y', e_1, e_2)$ are covered by a single symbolic superposition run.

5.2 Generating symbolic test suites

The problem of generating a symbolic test suite can be viewed as a symbolic state space exploration problem in which we search 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 computes a set of symbolic test cases of form $\langle \omega, G \wedge H \rangle$, such that there is a symbolic run of the EFSM over ω with path condition G which symbolically covers an accepting symbolic observer state $\mathfrak{q}_f$ provided H. The generated set can be controlled by varying parameters *stopcond* and *acceptcond*. A natural instantiation of these parameters will make sure that each conditioned observer state $\langle \mathfrak{q}_f, H \rangle$ for which there is a covering symbolic run, will also be covered by some symbolic test case in the test suite.

In the algorithm of Figure 5.2

- Wait contains the symbolic superpositioned states waiting to be explored,
- Cov is the set of already covered accepting conditioned observer states,
- TS is the current set of generated symbolic test cases,
- L^{stop} is the set of stop locations in the EFSM, and
- *acceptcond* is a condition that determines whether a test case should be accepted for inclusion in a test suite. A natural such condition is that a test case should be included if it includes at least one accepting conditioned observer state which has not been previously generated; this can be expressed as $(\mathcal{Q} \cap \mathcal{Q}_f) \setminus \text{Cov} \neq \emptyset$.
- *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 Wait$= \emptyset$.

Initially, the set of already explored states is empty and the only state waiting to be explored is $\langle \langle l_0, \bullet, \mathtt{true} \rangle \parallel \{\langle \mathfrak{q}_0, \mathtt{true} \rangle\} \parallel \epsilon \rangle$, The algorithm then repeatedly examines symbolic superpositioned states from Wait, one at a time. A location in the EFSM with no outgoing edges is a stop location. For each selected symbolic superpositioned state, not a stop location, all successor states reachable from $\langle \langle l, \sigma, G \rangle \parallel \mathcal{Q} \parallel \omega \rangle$ via a

Cov := $\emptyset$, TS := $\emptyset$,
WAIT := $\{\langle\langle l_0, \bullet, \texttt{true}\rangle \parallel \{\langle \mathfrak{q}_0, \texttt{true}\rangle\} \parallel \epsilon\rangle\}$
while $\neg$*stopcond* **do**
  select_and_remove $\langle\langle l, \sigma, G\rangle \parallel \mathcal{Q} \parallel \omega\rangle$ **from** WAIT
  if $l \notin L^{stop}$ **then**
    for all $\langle\langle l, \sigma, G\rangle \parallel \mathcal{Q} \parallel \omega\rangle \overset{a(\overline{p})/\overline{\mathfrak{b}}}{\rightsquigarrow} \langle\langle l', \sigma', G'\rangle \parallel \mathcal{Q}' \parallel \omega \cdot a(\overline{p})/\overline{\mathfrak{b}}\rangle$
      add $\langle\langle l', \sigma', G'\rangle \parallel \mathcal{Q}' \parallel \omega \cdot a(\overline{p})/\overline{\mathfrak{b}}\rangle$ **to** WAIT
  else if *acceptcond* **then**
    TS := $\{\langle\omega, G \wedge H\rangle | \langle \mathfrak{q}, H\rangle \in (\mathcal{Q} \cap \mathcal{Q}_f)\} \cup$ TS
    Cov := $(\mathcal{Q} \cap \mathcal{Q}_f) \cup$ Cov
return TS

Figure 5.2. A symbolic state space exploration algorithm for test suite generation.

symbolic computation step are inserted into WAIT. A new symbolic test case is generated when reaching a stop location, but only if the accept condition, *acceptcond*, becomes true. The algorithm terminates when the stop condition, *stopcond*, becomes true.

A few more observations can be made on the algorithm in Figure 5.2:

- TS is a set, i.e., duplicates are removed, and
- Cov collects accepting symbolic observer states without considering the superposition condition H and without considering that two symbolic observer states including symbolic parameters may (later) evalute to the same coverage item.

It can here be noted that in the rest of this thesis we will only consider the case when H is `true`. This must be considered the most common case and for the observer predicates defined in this thesis, only those operating on guards (Section 4.1.1) may possibly require a superposition condition H different from `true`. Further, we will only consider the case when the symbolic observer states does *not* include symbolic parameters. Again, this must be considered the most common case, but if symbolic parameters are included this may cause the algorithm to include additional, unnecessary test cases for coverage, to the generated test suite.

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 deep-first algorithm consume less memory, but a biased left-to right search will cause a corresponding exploration of the search space.

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. 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 deep-first algorithm can be expected to peform best when generating a test suite. But the selection from WAIT (by **select_and_remove** in Figure5.2) is significant. Thus, a random selection can be expected to perform better than a biased left-to right selection. Still better, if possible, is to use some algorithm to efficiently approximate a best-first selection.

A coverage criterion gives rise to a (often large) set of coverage items, of which several may not be feasible. For a given coverage criterion, the test suite generation can be considered to incur a *cost*, measured in

- *time* to generate or execute the test suite, or
- *size* of the test suite where the size can be e.g., the number of coverage items covered, number of tests cases, or the total length (in computation steps) of all symbolic test cases.

It might therefore be desirable to optimize test suite generation for specific purposes. We will here just briefly illustrate with two examples on how the search exploration algorithm in Figure 5.2 can be modified.

Example 5.3 Assume we are interested to optimize on the number of coverage items covered. For some coverage criterion and EFSM it may be beneficial to modify the search exploration algorithm to additionally support removing test cases from TS. Define *acceptcond* as $(\{\mathfrak{q} | \langle \mathfrak{q}, H\rangle \in \mathcal{Q}, \mathfrak{q} \in \mathcal{Q}_f\}) \setminus \mathsf{Cov} \neq \emptyset$. Thus, test cases are accepted for inclusion in the generated test suite if covering at least one coverage item, not covered by any (previously generated) test case. Further, assume we have generated test cases in TS such that Cov is

$$\{\langle du('Y', e_0, e_2), \mathtt{true}\rangle, \langle du('X', e_1, e_3), \mathtt{true}\rangle\}$$

when another test case contributes with the coverage items

$$\{\langle du('Y', e_0, e_2), \mathtt{true}\rangle, \langle du('X', e_1, e_3), \mathtt{true}\rangle, \langle du('Y', e_2, e_3), \mathtt{true}\rangle\}$$

then all the existing test cases in TS are superseded and can be removed. In general, all test cases whose corresponding coverage items forms a subset of the coverage items contributed by the new test case can be removed. The search exploration algorithm in Figure 5.2 can be modified

to remove superseded test cases by replacing the line

$$\mathsf{TS} := \{\langle \omega, G \wedge H\rangle | \langle \mathfrak{q}, H\rangle \in \mathcal{Q}, \mathfrak{q} \in \mathcal{Q}_f\} \cup \mathsf{TS}$$

with

$$\begin{aligned}\mathsf{TS} := \ & \textbf{update TS with } \{\langle \omega, G \wedge H\rangle | \langle \mathfrak{q}, H\rangle \in \mathcal{Q}, \mathfrak{q} \in \mathcal{Q}_f\} \\ & \textbf{annotate } \{\mathfrak{q} | \langle \mathfrak{q}, H\rangle \in \mathcal{Q}, \mathfrak{q} \in \mathcal{Q}_f\}\end{aligned}$$

where all test cases in TS superseded by this new test case are removed and each test case added is annotated with coverage items covered.

□

Example 5.4 Assume we are interested to optimize on a test suite with the least weight as defined by a function

$$weight(\mathcal{Q}_i, cost(t_i))$$

where $\mathcal{Q}_i$ is the set of *new* coverage items contributed by test case t_i, that calculates how favorable a test case t_i is.

Define *acceptcond* as

$$\begin{aligned} & weight(\{\mathfrak{q} | \langle \mathfrak{q}, H\rangle \in \mathcal{Q}_i, \mathfrak{q} \in \mathcal{Q}_f\}, cost(\langle \omega_i, G_i \wedge H\rangle)) > MinWeight \\ \wedge \quad & (\{\mathfrak{q} | \langle \mathfrak{q}, H\rangle \in \mathcal{Q}_i, \mathfrak{q} \in \mathcal{Q}_f\}) \setminus \mathsf{Cov} \neq \emptyset \end{aligned}$$

where the calculated weight must be the same for all superposition conditions H in $\mathcal{Q}_i$ (because of the algorithm definition), and $MinWeight$ is the least acceptable weight for inclusion in the test suite. Thus, a test case that contributes with many new coverage items and where $cost(t_i)$ returns a low value is more favorable than a test case that contributes with few new coverage items to the test suite and where $cost(t_i)$ returns a large value.

The search exploration algorithm in Figure 5.2 can be modified to remove superseded test cases by replacing the line

$$\mathsf{TS} := \{\langle \omega, G \wedge H\rangle | \langle \mathfrak{q}, H\rangle \in \mathcal{Q}, \mathfrak{q} \in \mathcal{Q}_f\} \cup \mathsf{TS}$$

with

$$\begin{aligned}\mathsf{TS} := \ & \textbf{update TS with } \{\langle \omega, G \wedge H\rangle | \langle \mathfrak{q}, H\rangle \in \mathcal{Q}, \mathfrak{q} \in \mathcal{Q}_f\} \\ & \textbf{annotate } weight(\{\mathfrak{q} | \langle \mathfrak{q}, H\rangle \in \mathcal{Q}_i, \mathfrak{q} \in \mathcal{Q}_f\}, cost(\langle \omega_i, G_i \wedge H\rangle)\end{aligned}$$

where all test cases in TS superseded by this new test case are removed and each test cases added is annotated with its weight.

□

5.3 Efficiently representing sets of states

A potential problem of test suite generation algorithms is to efficiently represent and manipulate the sets $\mathcal{Q}$, 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.

We will here restrict discussion to the case in which all occurring superposition conditions H in a symbolic test case are `true`. Furthermore, we assume that symbolic observer states never depend on symbolic parameters, so that each symbolic observer state $\iota(\mathfrak{e}_1, \ldots, \mathfrak{e}_m)$ can be written as an observer state $\iota(d_1, \ldots, d_m)$ where $d_1, \ldots, d_m$ are values.

Recall that L^o is the set of observer locations. For an observer location $\iota \in L^o$, let $D_1, \ldots, D_m$ be the domains of the parameters $d_1, \ldots, d_m$ in observer states of form $\iota(d_1, \ldots, d_m)$. A set $\mathcal{Q}$ of conditioned observer states can then be represented as a mapping

$$\mathcal{Q} : L^o \rightarrow (D_1 \times \cdots \times D_m \rightarrow [0, 1])$$

such that for each observer location ι, we have $\mathcal{Q}(\iota)(d_1, \ldots, d_m) = 1$ if $\langle \iota(d_1, \ldots, d_m), \mathtt{true} \rangle \in \mathcal{Q}$, otherwise $\mathcal{Q}(\iota)(d_1, \ldots, d_m) = 0$. In a symbolic superpositioned computation step of form

$$\langle \langle l, \sigma, G \rangle \parallel \mathcal{Q} \parallel \omega \rangle \overset{a(\overline{p})/\overline{\mathfrak{b}}}{\rightsquigarrow} \langle \langle l', \sigma', G' \rangle \parallel \mathcal{Q}' \parallel \omega \cdot a(\overline{p})/\overline{\mathfrak{b}} \rangle$$

this mapping is then updated according to

$$\mathcal{Q}'(\iota')(d'_1, \ldots, d'_{m'}) := \bigvee_{\langle \iota(d_1, \ldots, d_m), \mathtt{true} \rangle \overset{\varsigma^s}{\hookrightarrow} \langle \iota'(d'_1, \ldots, d'_{m'}), \mathtt{true} \rangle} \mathcal{Q}(\iota)(d_1, \ldots, d_m) \ ,$$

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, i.e., the current symbolic annotated computation step of the EFSM, represented as match variables.

Now, recall the definition of the conditioned observer step relation, from Section 3.8, of form

$$\langle \iota(d_1, \ldots, d_m), \mathtt{true} \rangle \overset{\varsigma^s}{\hookrightarrow} \langle \iota'(d'_1, \ldots, d'_{m'}), \mathtt{true} \rangle$$

that implies that for some ς^s there is a transition in the operational semantics of form

$$\langle \iota(d_1, \ldots, d_m), \varsigma^s, \bullet \rangle \Rightarrow \langle \{\mathtt{next\_state}, \iota'(d'_1, \ldots, d'_{m'})\}, \mathtt{true}, \epsilon, \varsigma^s, \rho \rangle \ ,$$

which happens only if there exists an observer edge clause of form

$$\iota(u_1, \ldots, u_m) \ \mathtt{when} \ h \ \mathtt{->} \ oce$$

such that $d_1, \ldots, d_m$ match the patterns $u_1, \ldots, u_m$, the observer guard h is satisfied by the symbolic annotated computation step, represented by ς^s, and $\texttt{\{next\_state,}\iota'\texttt{(}d'_1\texttt{,}\ldots\texttt{,}d'_{m'}\texttt{)\}}$ is the resulting value of evaluating *oce*. We can summarize this discussion by rewriting the rule for updating the set $\mathcal{Q}$ as

$$\mathcal{Q}'(\iota')(d'_1, \ldots, d'_{m'}) := \bigvee_{\substack{\iota(u_1,\ldots,u_m)\ \texttt{when}\ h\ \texttt{->}\ oce \\ match\,(u_1,\ldots,u_m,d_1,\ldots,d_m,\bullet,\rho') \\ \langle h, \varsigma^s, \rho'\rangle \Rightarrow \langle \texttt{true}, \epsilon, \varsigma^s, \rho''\rangle \\ \langle oce, \varsigma^s, \rho''\rangle \Rightarrow \langle \texttt{next\_state,}\iota'\texttt{(}d'_1\texttt{,}\ldots\texttt{,}d'_{m'}\texttt{)}, \epsilon, \varsigma^s, \rho'''\rangle}} \mathcal{Q}(\iota)(d_1, \ldots, d_m) \ ,$$

where we have simplified the condition for updating $\mathcal{Q}'(\iota')(d'_1, \ldots, d'_{m'})$ by omitting preceding observer edge clauses, not matching, h not evaluating to `true`, or *oce* not evaluating to $\texttt{next\_state,}\iota'\texttt{(}d'_1\texttt{,}\ldots\texttt{,}d'_{m'}\texttt{)}$.

Example 5.5 Lets take a closer look on how the observer states in the Definition-Use pair observer (Section 4.1.4) are affected by an conditioned observer step. Let V be the set of state variables and E the set of edges in an EFSM. We then have that,

$$\begin{aligned} & \left\{ \langle q_0(), \texttt{true}\rangle \right\} \\ \cup\ & \left\{ \langle q_1(x,e), \texttt{true}\rangle \mid x \in V \ \wedge\ e \in E \right\} \\ \cup\ & \left\{ \langle du(x,e,f), \texttt{true}\rangle \mid x \in V \ \wedge\ e, f \in E \right\} \end{aligned}$$

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.

$$\mathcal{Q}(q_0) := 1 \tag{5.1}$$

$$\mathcal{Q}'(q_1)(x,e) := \left(\begin{array}{ll} & \left(\begin{array}{ll} & \mathcal{Q}(q_0) = 1 \\ \wedge & \texttt{is\_definedvar(}x\texttt{)} \\ \wedge & \texttt{is\_edge(}e\texttt{)} \end{array} \right) \\ \vee & \left(\begin{array}{ll} & \mathcal{Q}(q_1)(x,e) = 1 \\ \wedge & \neg\texttt{is\_definedvar(}x\texttt{)} \end{array} \right) \end{array} \right) \tag{5.2}$$

$$\mathcal{Q}'(du)(x,e,f) := \left(\begin{array}{ll} & \left(\begin{array}{ll} & \mathcal{Q}(q_1)(x,e) = 1 \\ \wedge & \texttt{is\_usedvar(}x\texttt{)} \\ \wedge & \texttt{is\_edge(}e\texttt{)} \end{array} \right) \\ \vee & \left(\begin{array}{ll} & \mathcal{Q}(du)(x,e,f) = 1 \\ \wedge & \texttt{true} \end{array} \right) \end{array} \right) \tag{5.3}$$

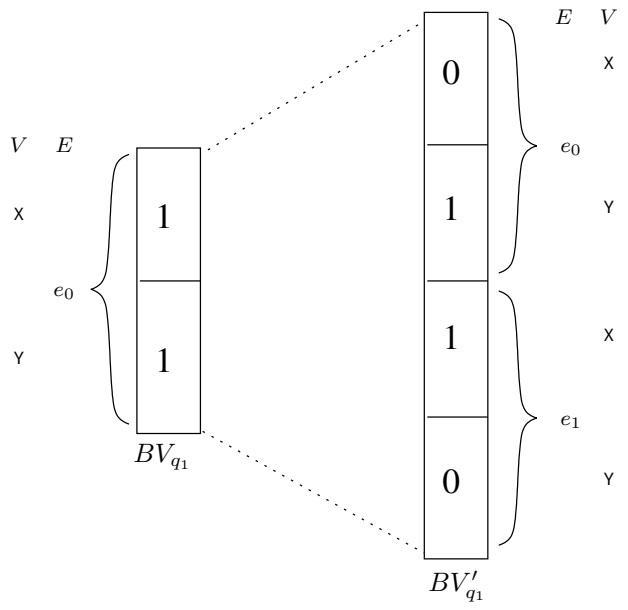


Figure 5.3. The bitvector BV_{q_1} represents the set of observer states $\{q_1(\mathsf{X}, e_0), q_1(\mathsf{Y}, e_0)\}$ before a conditioned observer step. As the EFSM holds a definition of a variable X on a (new) edge e_1, the size of the bitvector is adjusted, before Definition 5.2 (from Example 5.5) is applied. The bitvector BV'_{q_1}, after the conditioned observer step, then represents the set of observer states $\{q_1(\mathsf{Y}, e_0), q_1(\mathsf{X}, e_1)\}$.

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.

□

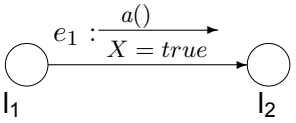
5.3.1 Bitvector Representation of Sets of Observer States

In our implementation, each mapping $\mathcal{Q}(\iota)$ is represented by a one-dimensional bitvector $BV_\iota : [1, \ldots, \mathfrak{M}_\iota] \to [0, 1]$, where $\mathfrak{M}_\iota = \prod_{1 \leq i \leq n} |D_i|$. By defining a bijection $f_\iota : (D_1 \times \cdots \times D_m) \to \mathfrak{M}_\iota$, we can represent $\mathcal{Q}(\iota)(d_1, \ldots, d_n)$ as $BV_\iota(f_\iota(d_1, \ldots, d_n))$. Since the size of the domains for $d_1, \ldots, d_n$ may not be known in advance we allow $\mathfrak{M}_\iota$ 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, but 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 $f_\iota(d_1, \ldots, d_n)$, 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 5.2, 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} represents the set of observer states $\{q_1(\mathsf{X}, e_0), q_1(\mathsf{Y}, e_0)\}$. 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 $\{q_1(\mathsf{Y}, e_0), q_1(\mathsf{X}, e_1)\}$. In Example 5.2 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 $\mathfrak{M}_{du} = 2 \times 3 \times 2 = 12$.

□

The contributions of coverage items for a symbolic test case is represented by the bitvectors for accepting observer locations after the last symbolic superpositioned computation step in a symbolic superposition run. Note that after each symbolic superposition run we may have contributions of coverage items from several accepting observer locations, each represented by a bitvector. We generate test suites that fulfill a given coverage criteria and include many symbolic test cases. Thus, we additionally need to maintain global bitvectors (one for each accepting observer location) where the collected contributions of coverage items from each symbolic superposition run can be represented. That is, after each symbolic superposition run accepted by the observer, we compare the global bitvectors with the bitvectors for the accepting observer locations. If the test case contributes with any new coverage item the test case is included in the test suite otherwise it is not.

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 $\overline{\phi}$ 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 $\overline{\phi}$ 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:

	MlpAge	*Clock*	*Age*
$\overline{d_1} =$	1	0	1
$\overline{d_2} =$	2	0	2
$\overline{d_3} =$	2	≤ 1	1
$\overline{d_2} =$	3	0	3
$\overline{d_4} =$	3	≤ 1	2
$\overline{d_5} =$	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 an abstract test case in Section 5.4.1.

To validate if output sent from the SUT conforms to an output event $b(\overline{d})$ in an abstract test case we must consider all concretizations of $b(\overline{d})$. We do this by an an *abstraction* of the output sent from the SUT to $b(\overline{d})$. 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.

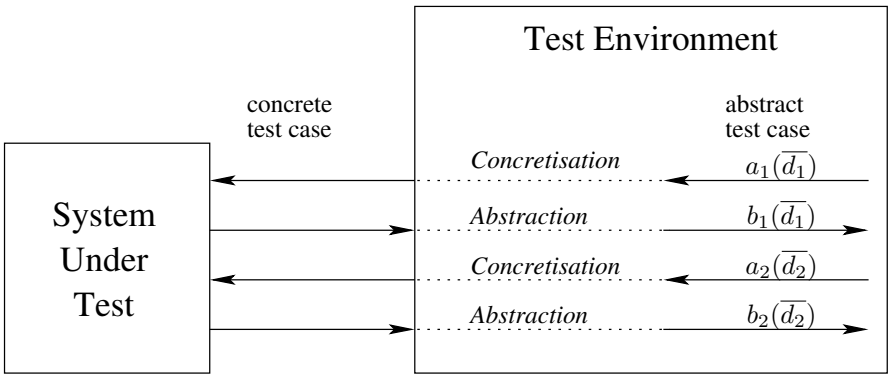


Figure 5.4. Outline of how input events in an abstract test cases are made concrete before sent to the SUT, and responses from the SUT made abstract before validating.

It can be noted that both concretisation and abstraction may depend on the abstract test case. Thus, for example, in Figure 5.4, $a_1(\overline{d_1})$ and $a_2(\overline{d_2})$ may be exactly identical abstract events (i.e., $a_1 = a_2$ and $\overline{d_1} = \overline{d_2}$), 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 concretizations 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.

The algorithm in Figure 5.2 can be modified to support filter usage using an accept condition (*acceptcond*) set to $\{\mathfrak{q}|\langle\mathfrak{q}, H\rangle \in \mathcal{Q}, \mathfrak{q} \in \mathcal{Q}_f\} \neq \emptyset$. 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 oberver 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.

□

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 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 Validating the 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/EFSM 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 (that is true when an accepting observer state is reached). To validate properties, expressible as abserver automata, the algorithm in Figure 5.2 needs modification of the accept condition (*acceptcond*) to $\{\mathfrak{q}|\langle \mathfrak{q}, H\rangle \in \mathcal{Q}, \mathfrak{q} \in \mathcal{Q}_f\} = \emptyset$ and if this condition is not fulfilled the algorithm terminates and returns `false`. If all symbolic symbolic superposition runs fulfills the accept condition `true` is returned. Further, as we do not keep track of coverage items or generate a test suite, Cov and TS can be removed. The resulting state search 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`.

$\textsc{Wait} := \{\langle\langle l_0, \varsigma_0^s, \texttt{true}\rangle \parallel \{\langle q_0, \texttt{true}\rangle\} \parallel \epsilon\rangle\}$
while $\textsc{Wait} \neq \emptyset$ **do**
  select_and_remove $\langle\langle l, \sigma, G\rangle \parallel \mathcal{Q} \parallel \omega\rangle$ **from** $\textsc{Wait}$
  if $l \notin L^{stop}$ **then**
    for all $\langle\langle l, \sigma, G\rangle \parallel \mathcal{Q} \parallel \omega\rangle \overset{a(\overline{p})/\overline{\mathfrak{b}}}{\rightsquigarrow} \langle\langle l', \sigma', G'\rangle \parallel \mathcal{Q}' \parallel \omega \cdot a(\overline{p})/\overline{\mathfrak{b}}\rangle$
      add $\langle\langle l', \sigma', G'\rangle \parallel \mathcal{Q}' \parallel \omega \cdot a(\overline{p})/\overline{\mathfrak{b}}\rangle$ **to** $\textsc{Wait}$
  else if $|\{\mathfrak{q} | \langle \mathfrak{q}, H\rangle \in \mathcal{Q}, \mathfrak{q} \in \mathcal{Q}_f\}| = 0$ **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 99b].

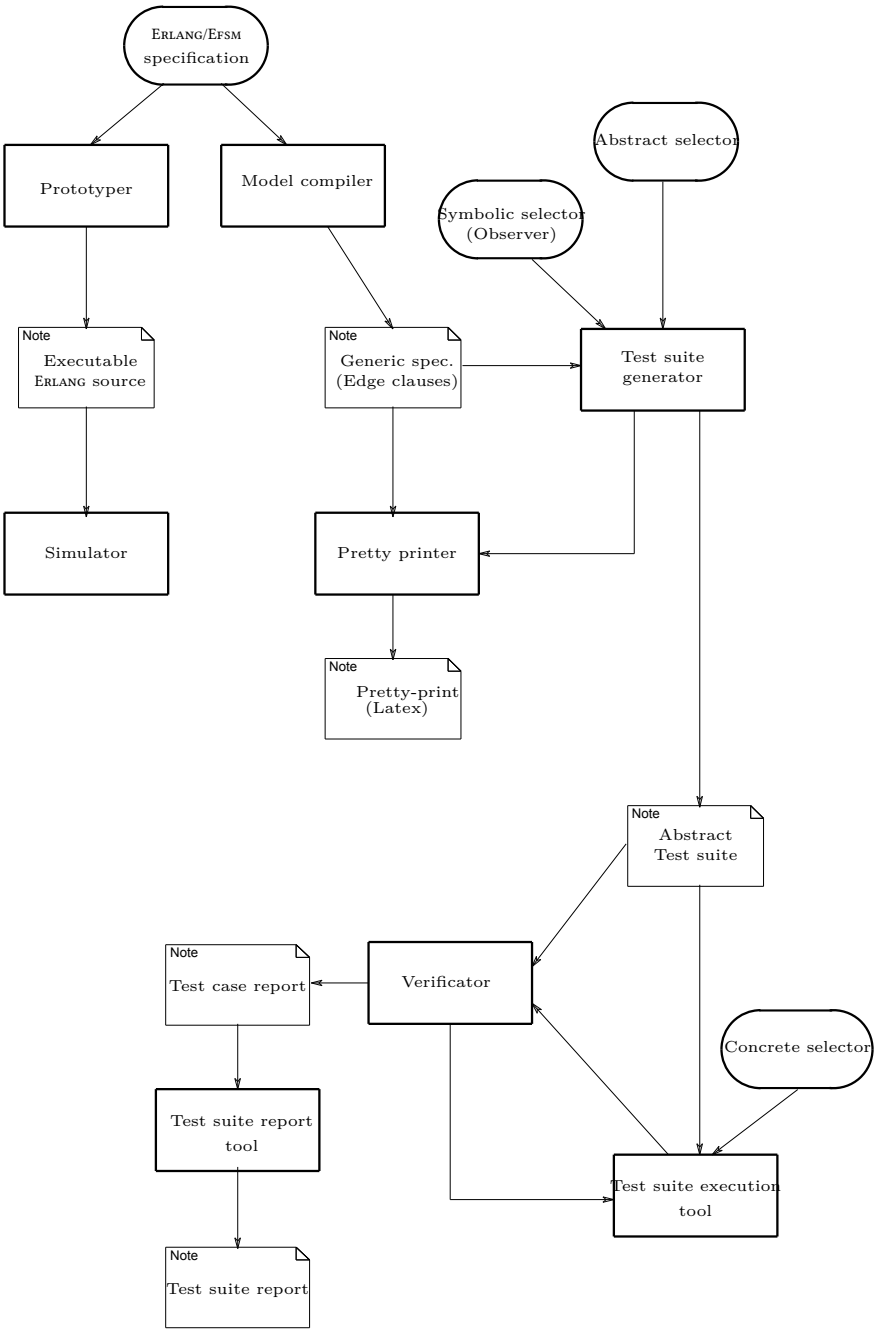


Figure 6.1. Overview of features in the Erly Marsh tool.

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 eactly 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) [E. 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 is distributed, we need a mechanism to disseminate these data settings to each component in the test execution environment. 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 curently 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 internally to distinguish between different test cases and it is assumed that the SUT can use existing parameters (e.g., a sequence number).

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/Efsm 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.

□

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 possbility 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 succesful 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).

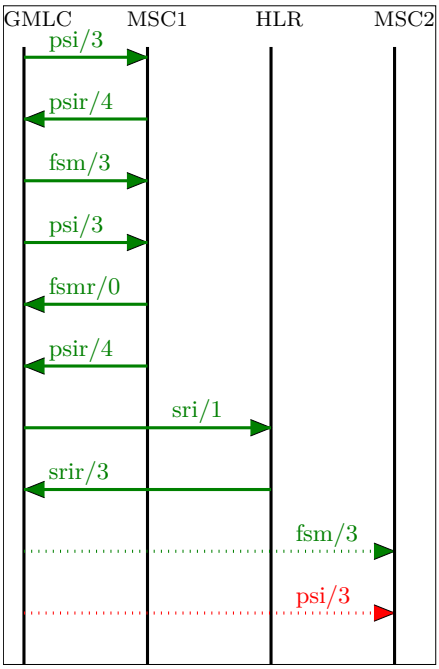


Figure 6.2. A screen dump from Erly Marsh after executing a test suite with failing test cases where `GMLC` is the SUT and `MSC1`, `MSC2` and `HLR` are parts of the test execution environment. The figure shows how 10 test cases form a cluster by sharing a trace (green solid lines), ending with a received *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 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/Efsm 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/LTE 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/LTE 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 Application is followed by a sequence of MAP requests to (optionally) force an update of cached presence information in VLR and HLR, followed by a MAP request to access the presence data.

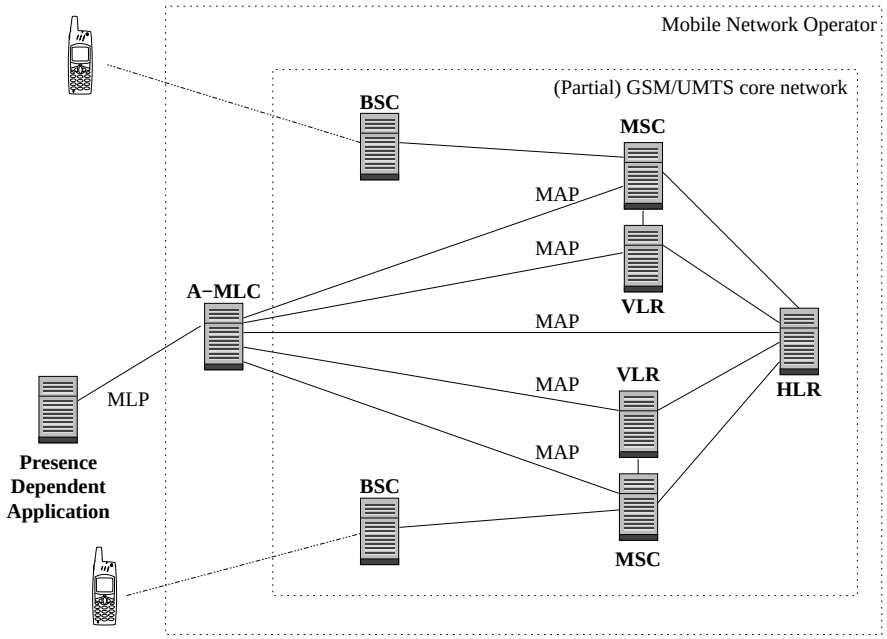


Figure 7.1. An outline of how Mobile Arts A-MLC can communicate with a presence dependent application and mobile devices using a number of GSM/UMTS/LTE core network nodes.

The implementation of A-MLC was made mainly in Erlang, utilizing Erlang OTP, with approximately 130,000 lines of Erlang code and 5,500 lines of C code. Development was made in a typical fashion by first creating a requirement specification. From the set of requirements, a detailed functional specification consisting of a textual description and a set of Message Sequence Diagrams was created. Finally, the implementation was based on the detailed functional specification. During this process the A-MLC was frequently updated, creating problems to keep the functional specification and implementation of the system consistent. As the environment demands high requirements on functional correctness, availability and fault tolerance, it was decided to formalize the functional specification and complement a manually created (hand-crafted) test suite with a test suite based on the formalization.

The process of creating the specification took a significant amount of effort over the complete period of time the evaluation lasted. During this process the specification was updated frequently, typically because an executed test suite revealed a failure that could be attributed to an error 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 exists 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 spec-

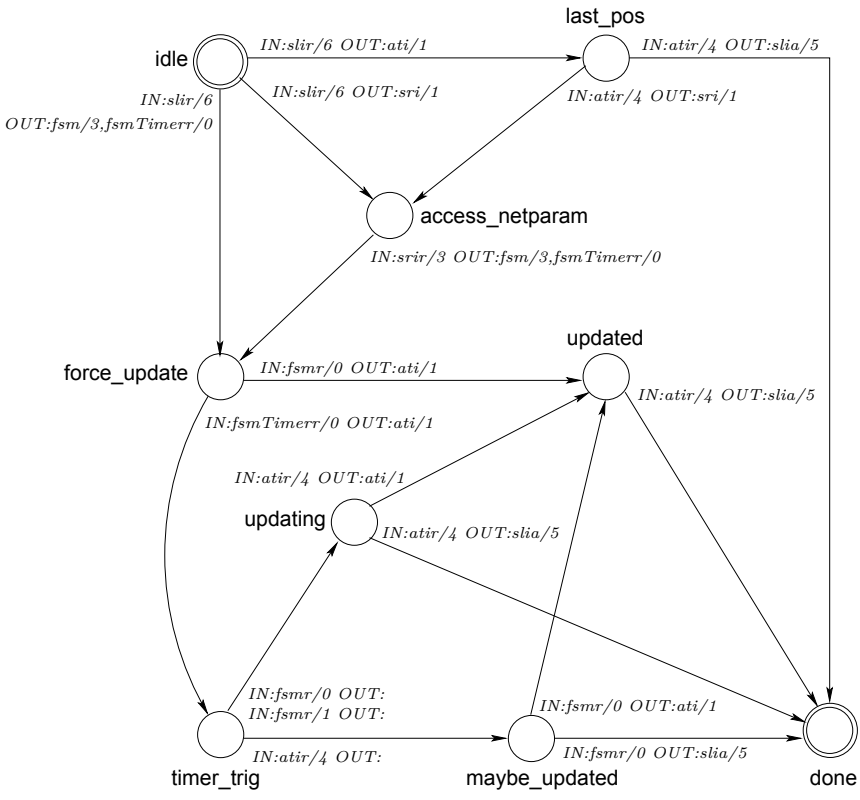


Figure 7.2. A graphical representation of a the part of the state machine created for A-MLC, covering the ATI presence method. Note that due to space restrictions, all lists of event parameters are simplified to a single parameter, which merely indicates whether the real parameters are MAP or MLP parameters. Further, some edges are omitted, and all state variable updates and guards on edges are not shown.

ification 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,

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/LTE 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.

□

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/LTE 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

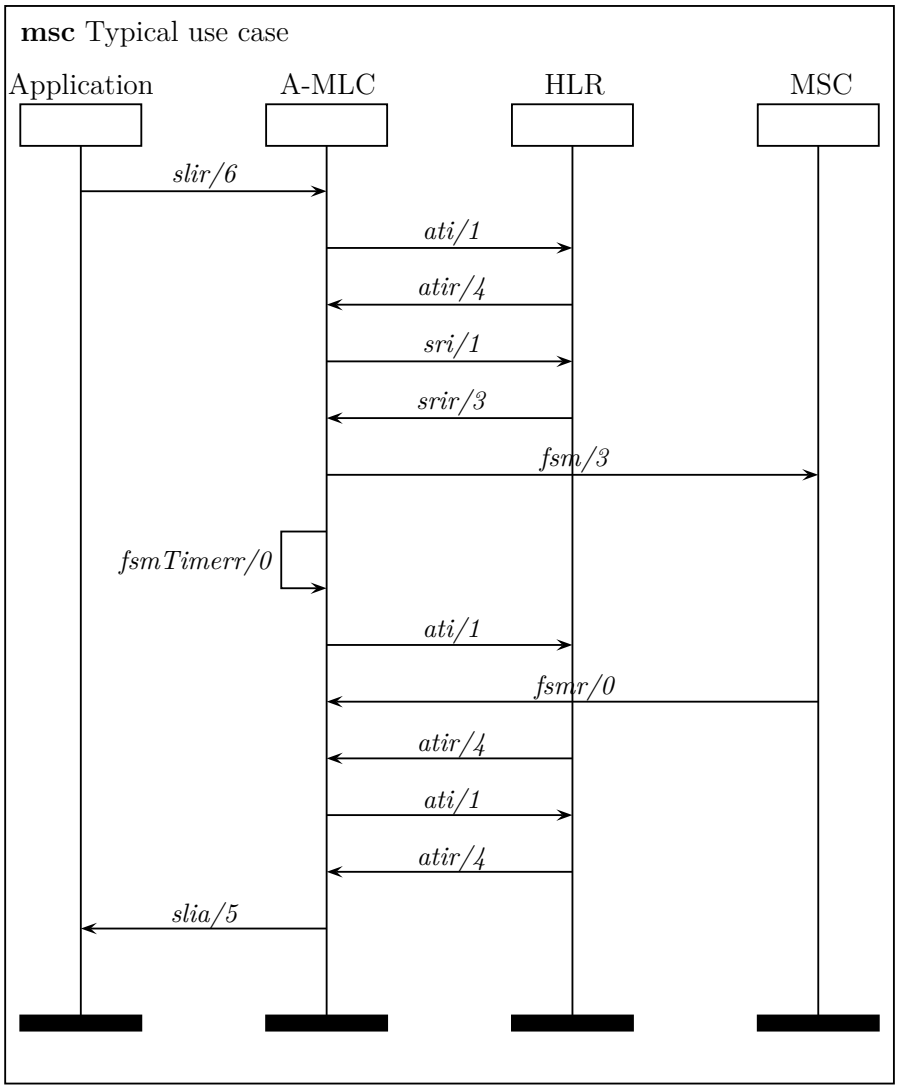


Figure 7.3. Typical execution of a test case in which execution starts by sending a *slir/6* request to the A-MLC, the A-MLC then executes a number of SS7 MAP operations (i.e., *ati/1*, *sri/1* and *fsm/3*) and returns a *slia/5* response. Note that due to space restrictions, all event parameter lists are simplified to a single parameter only indicating if real parameters are MAP or MLP parameters.

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.

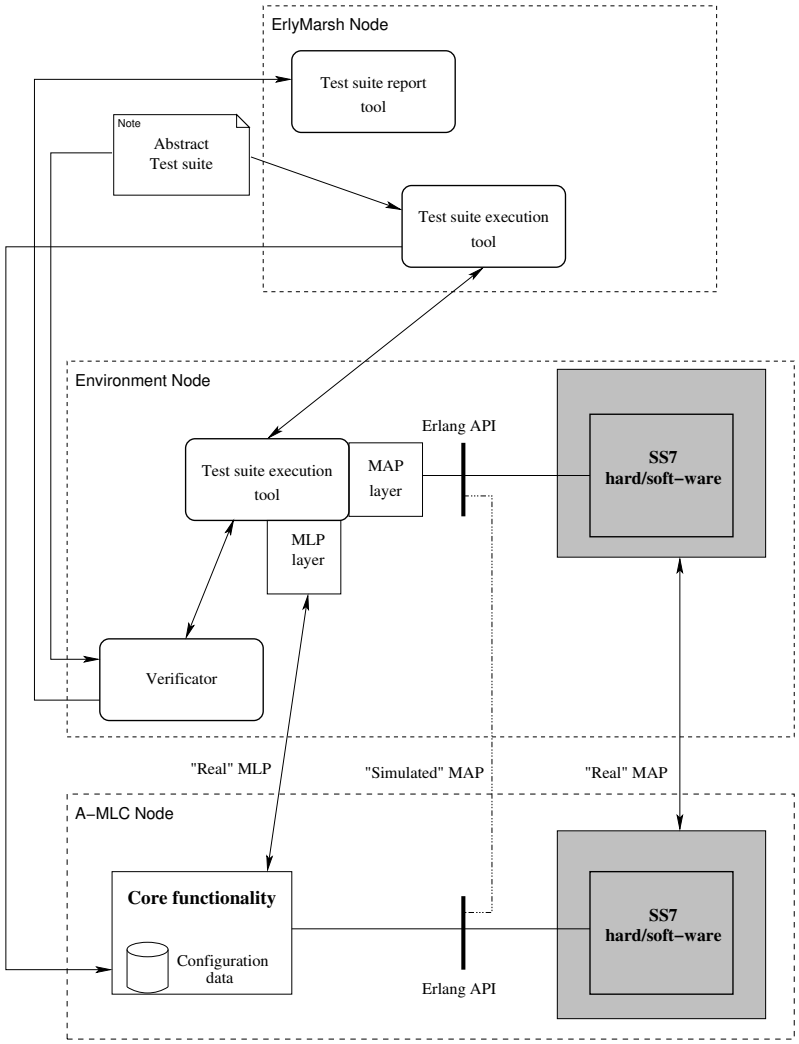


Figure 7.4. Overview of the test execution environment for the A-MLC product. The "grey" parts were dedicated SS7 hard/soft-ware not used in this evaluation.

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.
- 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 execu-

tion 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 Verificator 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/Efsm, 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/Efsm 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.

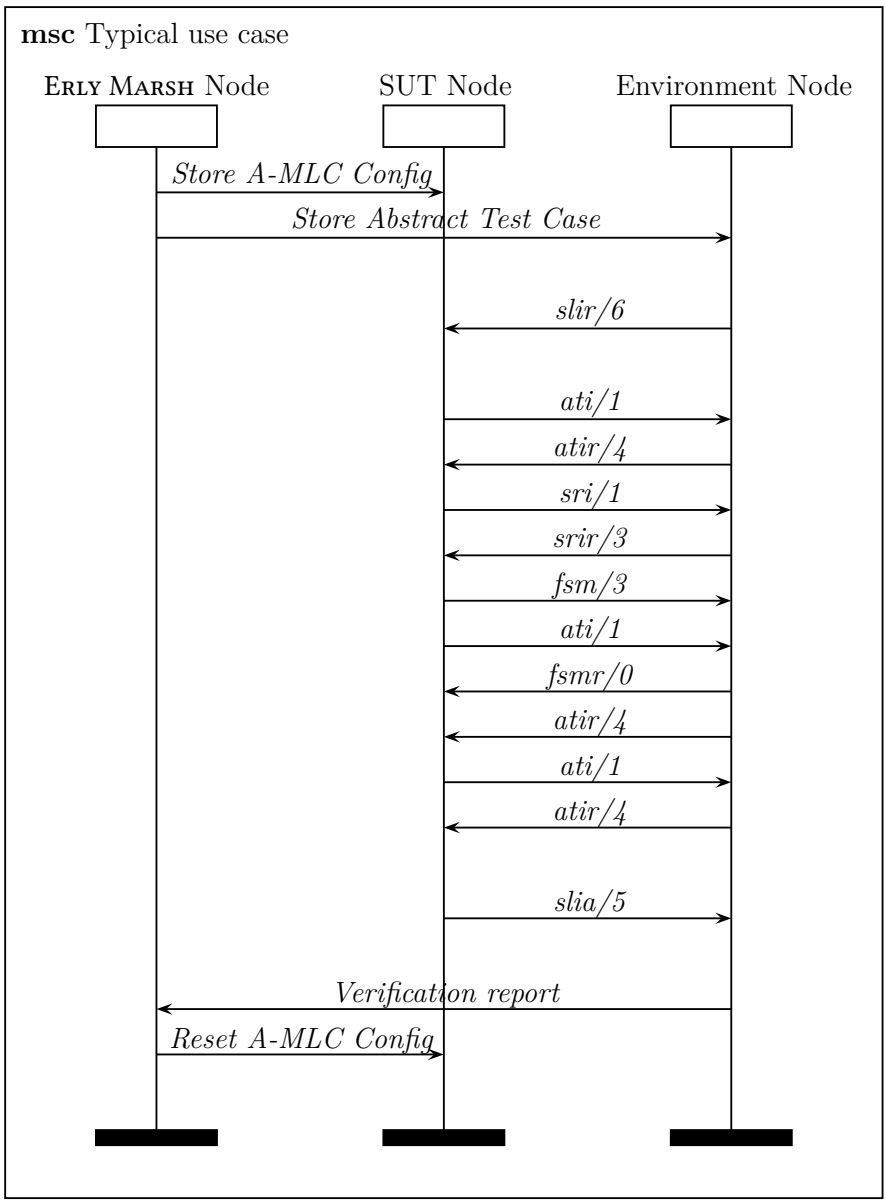


Figure 7.5. Typical execution of a test case in a simulated environment where execution start by configure A-MLC with configuration data and the abstract test case in the Test suite execution tool and Erly Marsh Verificator in the Environment Node. The test case is then initiated by sending a *slir/6* 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.

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 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 use the formula

$$W^{ew} = W^{tcw}\frac{P}{T}$$

where

- W^{ew} is the event weight.
- W^{tcw} is the test case weight.

- P is the probability for event types in a symbolic trace to be selected. For example, P becomes $\frac{1}{10} \times \frac{1}{10} \times \frac{1}{10} = 10^{-3}$ if there exists 10 possible types of events, for a symbolic trace with 3 input expressions.
- T is the calculated maximum number of different abstract test cases a symbolic test case represents, without considering any constraints given by guard expressions. For example, T becomes $10 \times 10 \times 10 = 10^3$ if there exists 3 symbolic parameters in a test case, all with a domain of size 10,

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 does not have constraints on all parameters. This tends test case weighting to favor longer test cases. Event weighting on the other hand, favor short test cases as T becomes larger and P 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* 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/Efsm 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 trans-

lated 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, All-Edges^{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 purpose 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 my 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 $s_1, \ldots, s_n$ 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.

□

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 [Eri 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 elliminate 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 requires 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/LTE 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 present the results of applying the framework for model-based testing, introduced in this thesis, on A-MLC. 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 our findings. Section 8.2 further discusses the results of measuring failures, Section 8.3 discusses the results of measuring faults, Section 8.4 discusses the results of measuring code coverage, Section 8.5 discusses the results of measuring size of test suites and execution times, and finally in Section 8.6 we summarize and discuss all results.

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,

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.*

- 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.

Test case weight:

Test suite name:	Rnd^{tcw}_{1k}	Rnd^{tcw}_{12k}	Rnd^{tcw}_{48k}	Rnd^{tcw}_{100k}
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^{ew}_{1k}	Rnd^{ew}_{12k}	Rnd^{ew}_{48k}	Rnd^{ew}_{100k}
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.*

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 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

We also created a Manual test suite from a previously created test suite when testing A-MLC without the use of Erly Marsh, see Section 7.2.2. 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. As a consequence, the test suite used at Mobile Arts for testing A-MLC was limited to 20 test cases. The test suite only covered 6 out of 11 traffic scenarios defined in the functional specification and mainly included test cases representing the most frequently occurring use cases. We will refer to this limited test suite as Man_{lim}. Manual test cases for the additional traffic scenarios, covering additional presence methods, were constructed by the author following the same patterns as in the Man_{lim} test suite. The full manual test suite is referred to as Man.

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 (fail_{10} and fail_{13}), 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:* $fail_1, ..., fail_5$. 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 types towards the GSM/UMTS/LTE core network occurred as unexpected events in some test case. As over utilization of the HLR should be avoided (most of the GSM/UMTS/LTE core network depends on the HLR) failures detected with an additional output event towards the HLR in the GSM/UMTS/LTE core network was classified as a middle severity failure. These were fail_2 (unexpected first *sri/1*), fail_3 (unexpected second *sri/1*), and fail_4 (unexpected *ati/1*). All other failures (fail_1 with unexpected *fsm/3* and fail_5 with unexpected *psi/3*) were classified as low severity failures.
- *Bad output:* $fail_6, ..., fail_7$. Test cases where the test execution environment expects to receive one output event type, but receives another output event type, i.e., the GSM/UMTS/LTE core network did not receive the expected output event. These failures occurred when an immediate position request was expected, but the A-MLC instead chose to first force an update of the position (fail_6 low sever-

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.*

ity, expected *psi/3* but got *fsm/3*) or request additional information from the HLR ($fail_7$ middle severity, expected *psi/3* but got *sri/1*).

- *Missing output:* $fail_8, ..., fail_{11}$. Test cases 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/LTE 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.

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$^{tcw}_{48k}$ and Rnd$^{tcw}_{100k}$), 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. But it did not find the highly critical failure, fail$_{14}$, as that particular failure 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. These faults can be classified by their origin as:

- Functional faults are faults that were found by comparing the expected behavior in the formal model with the actual behavior of the SUT.
- Resource faults are 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 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 than 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,

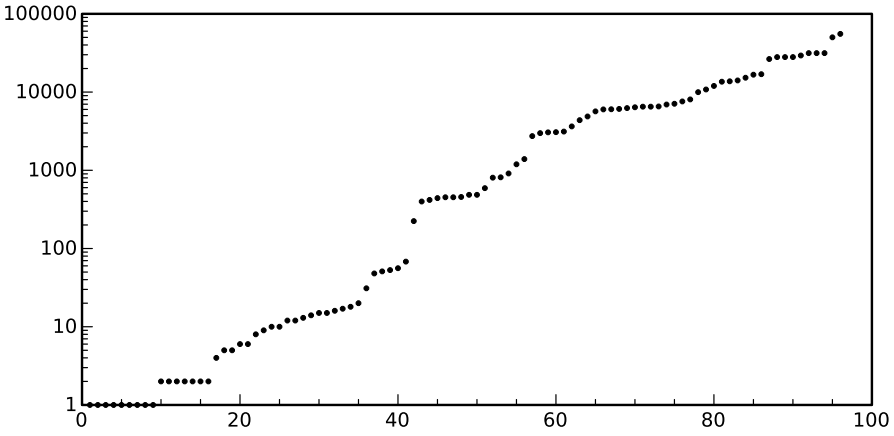


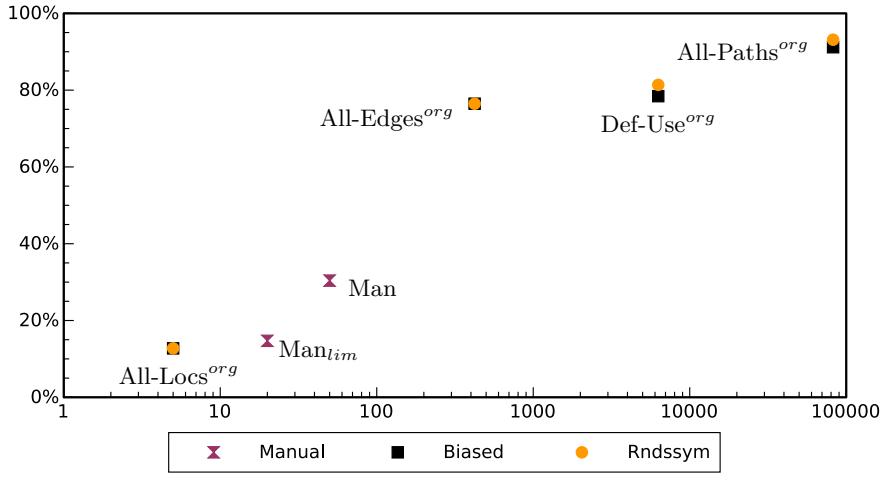
Figure 8.1. Number of test cases exposing each bug fix when running the All-Paths^{org} test suite with random abstract test case selection. The x-axis shows the number of test cases exposing a bug fix, and the y-axis shows the number of different bug fixes exposed by the same number of test cases.

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 the chosen base model had an effect of the outcome, although limited. Test suites selected from the original base model in general perform better than the normalized base model. Random abstract test case selection (Rndsym) also perform better than biased (Biased) regardless of base model. Also, the Man test suite did expose relatively few bug fixes.

Next we ran random test suites using test case weights (Rnd^{tcw}) and event weights (Rnd^{ew}). As can be seen in Figure 8.3 the coverage-based test suites, with the exception of All-Locs, performed rather well. We note that the random test suite using a uniform distribution over all 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 is to be expected since its bias will leave some parts of the model unexplored. 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. 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

Original base model:



Normalized base model:

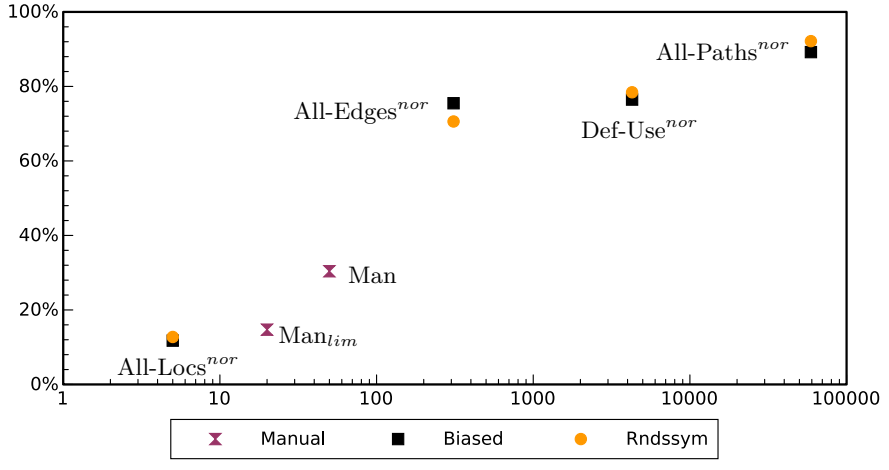


Figure 8.2. Relative number of different bug fixes exposed when running the coverage based test suites. For reference, the manual test suites are additionally included. For both the original and normalized base model the x-axis shows the number of test cases executed, and the y-axis shows the percentage of bug fixes (of all 102) that were exposed.

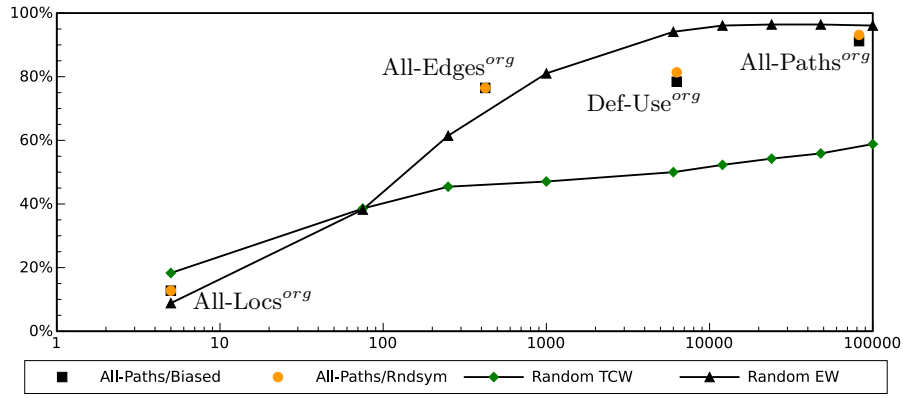


Figure 8.3. Relative number of different bug fixes exposed when running the coverage based test suites with the original base model and biased and random abstract test case selection, random test suites generated using test case weights (Rnd^{tcw}_{100k}), and random test suites generated using event weights (Rnd^{ew}_{100k}). 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.

suites increase, the law of decreasing marginal utility sets in; larger test suites only give a small improvement in the number of exposed faults.

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.

8.3.1 Characteristics of a selection of test suites

The tested version of A-MLC was a deployed version on which some further development was made. 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 the number of different bug fixes exposed by each test case part of the All-Paths^{org}, Rnd^{tcw}_{100k}, and Rnd^{ew}_{100k} test suites.

It can be noted that the Rnd^{tcw}_{100k} 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

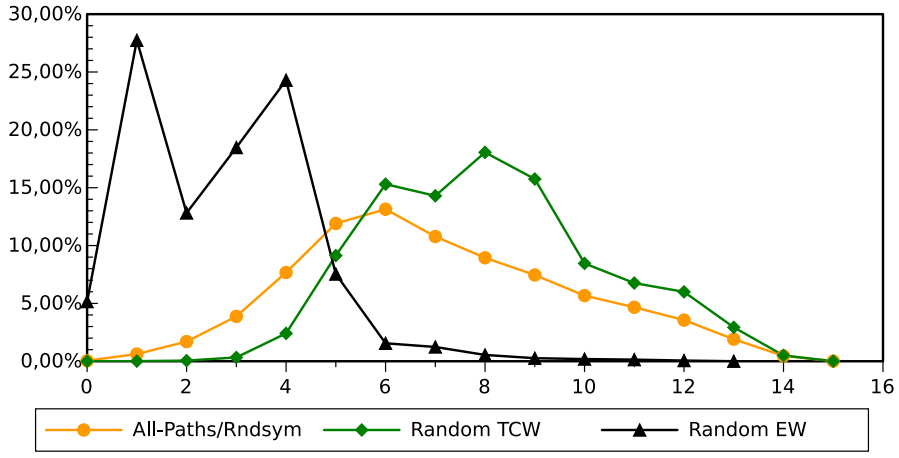


Figure 8.4. Number of different bug fixes exposed by test cases when running the All-Paths^{org} test suite with random abstract test case selection, the random Rnd$^{ew}_{100k}$ test suite, and the random Rnd$^{tcw}_{100k}$ test suite. The x-axis shows the number of bug fixes exposed by a single test case, and the y-axis shows the percentage of test cases in the test suite that expose the same number of different bug fixes.

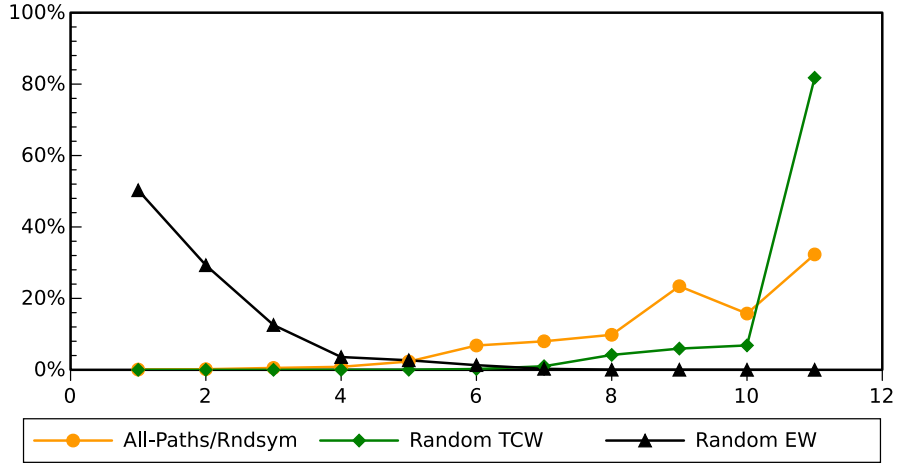


Figure 8.5. Length of test cases (number of edges) in the All-Paths^{org} test suite with random abstract test case selection, the random Rnd$^{ew}_{100k}$ test suite, and the random Rnd$^{tcw}_{100k}$ test suite. The x-axis shows the length of the test case, and the y-axis shows the fraction of test cases in the test suite with that length in the test suite.

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 shorter. The Rnd^{ew}_{100k} test suite, with many shorter test cases, has a high concentration of test cases requiring few bug fixes. 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. In the upper Figure 8.6 all test suites where executed in the same order as they were created, i.e., a random order with respect to the length of test cases. In the lower Figure 8.6 the test suites were sorted before execution such that the shortest test cases were executed first. As can bee seen, the results were quite different. In particular, 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 60000 test cases.

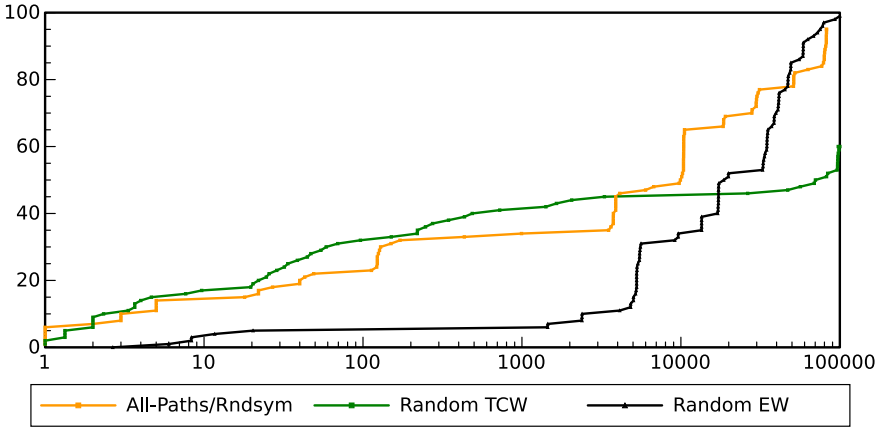
8.4 Code coverage

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.

As created:



Shortest first:

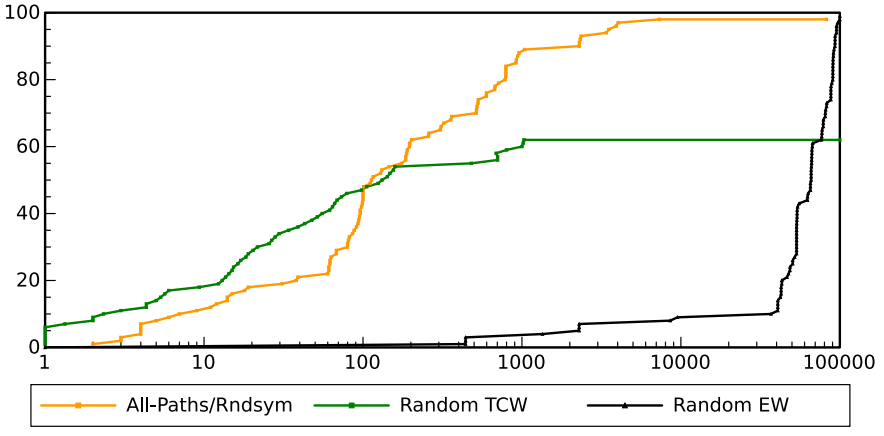


Figure 8.6. Accumulated bug fix usage when executing the test suites as created and sorted such that the shortest test cases are executed first. Results shown for the All-Paths^{org} test suite with random abstract test case selection, and two random test suites with 100,000 test cases using event weights (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.

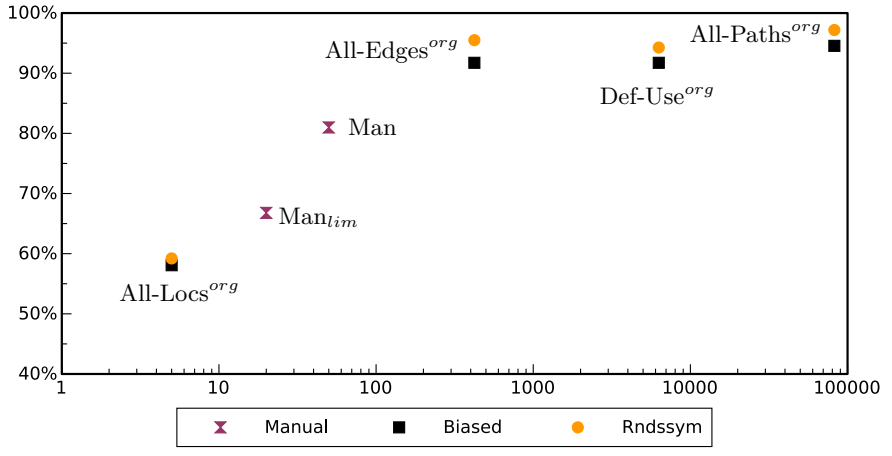
3. *Diagnostic functions:* Functions not used in normal traffic cases. Typically these are functions used for debugging, test, and operation and maintenance.
4. *Implemented behavior not modeled:* These are implemented features present in the SUT that has been excluded from the formal model, but still 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 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 Rndsym) we got the results in Table 8.6 for three main Erlang modules in the SUT.

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). Figure 8.9 shows the relation between exposed 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.

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.

Original base model:



Normalized base model:

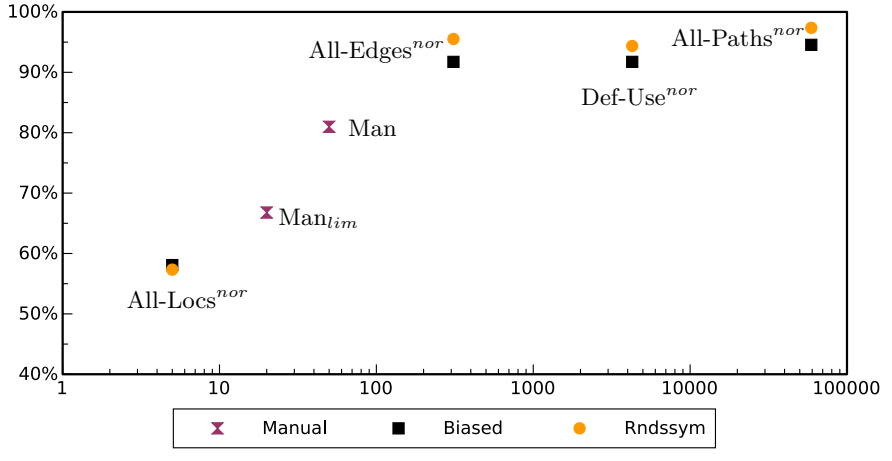


Figure 8.7. Relative source code coverage when running coverage based test suites. For reference, the manual test suites are additionally included. For both the original and normalized base model the x-axis shows the number of test cases executed and the y-axis shows the measured relative coverage.

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.*

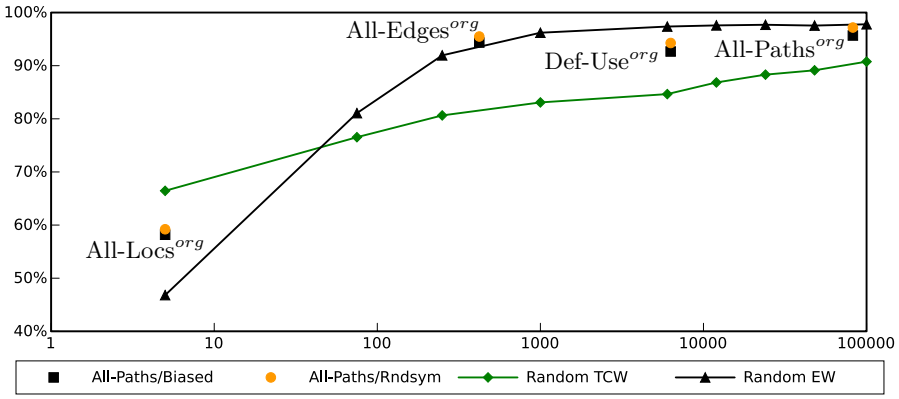


Figure 8.8. Relative source code coverage when running the random test suites and the coverage based test suites with the original base model. The x-axis shows the measured relative coverage and the y-axis shows the number of test cases executed.

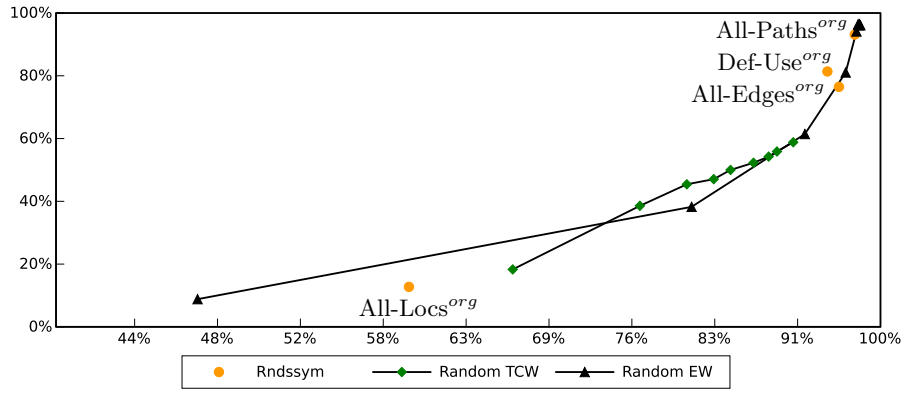


Figure 8.9. Relation between source code coverage and number of exposed bug fixes. with the original base model. The x-axis shows the measured relative coverage and the y-axis shows the percentage of exposed bug fixes.

8.5 Test suite size and execution times

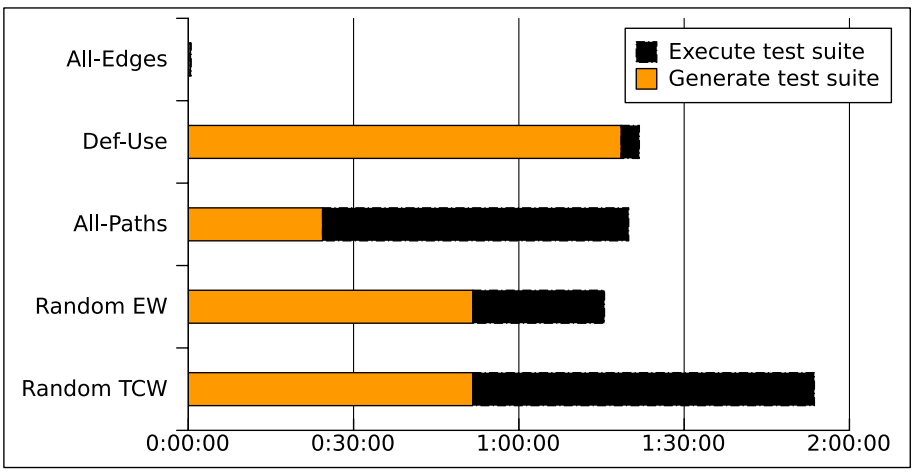


Figure 8.10. Comparison in time generating and concurrently executing the All-Edges^{org}, Def-Use^{org}, All-Paths^{org}, Rnd$^{tcw}_{100k}$, and Rnd$^{ew}_{100k}$ test suites.

Normally, in an industrial environment we only a strictly limited amount of time can be spent on testing. Thus, the usability of any test tool depends on whether a desired result can be returned after some given maximum amount of time.

Assuming an existing formal specification, we need to first generate the test suite, and then execute the test suite. Typically, during time, test suites must be generated much more seldom than executed. But as the experience with A-MLC shows, see Section 8.6.1, test suites may have to

be regenerated a significant number of times before arriving at a correct specification.

The amount of time spent generating and concurrently executing the coverage based test suites once is shown in Figure 8.10. Searching the base model is rather efficient, but the overhead of using a slightly more complicated observer, such as Def-Use^{org}, is rather significant with the current implementation of Erly Marsh. 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. 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.

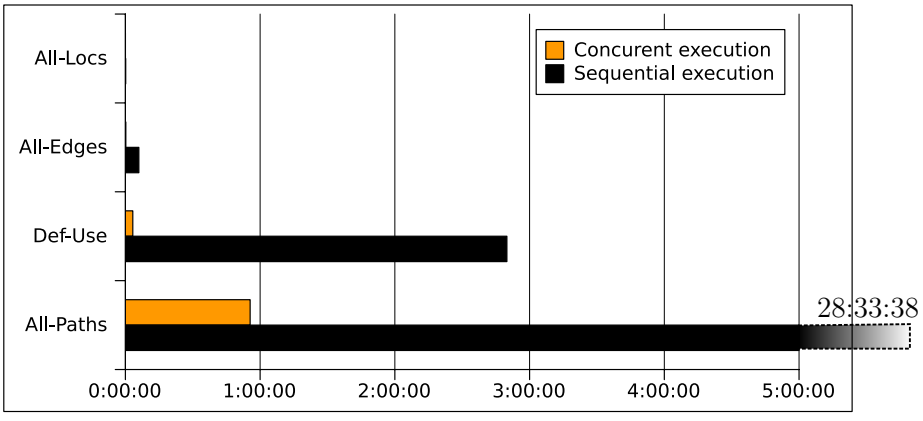


Figure 8.11. Comparing executing test suites concurrently and sequentially.

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

In order to evaluate our test suite generation techniques a case study of a commercially available telecom software system (Mobile Arts A-MLC) was performed. The case study was a time consuming activity with tools and formal specification evolving simultaneously. Mobile Arts A-MLC was implemented in Erlang which somewhat simplified the implementation of the test execution environment as we could use an Erlang API to simulate the MAP interface without using dedicated SS7 software/hardware that would otherwise been necessary.

The bulk of the work was clearly the creation of the formal model. The created Erlang/Efsm model of A-MLC became rather large because of a need to include many of the details in all the possible traffic scenarios. We then generated and compared different test suites with respect to failures detected, bug fixes exposed, source code coverage, and size of generated test suites. We have also measured execution times for generating and executing the test suites. The differences found between different test selection strategies are further discussed below.

On Coverage based testing: The coverage based test suites were executed on two different base models that gave slightly different results, despite 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 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. Thus, as we showed the abstract test case selection technique had an impact we are likely to miss more abstract test cases exposing bug fixes with less symbolic test cases, if only selecting a single abstract test case from each. 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. Improved coverage criteria and/or abstract test case selection techniques can possibly eliminate this difference. 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.

The influence a coverage based 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. In our evaluation, the random abstract test case selection (Rndsym) was measured to be more effective than the biased deterministic technique (Biased). This was as expected, but the differences were quite minimal though. Therefore, we did not investigate further possible improvements such as:

- 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.

The large number of test cases that the A-MLC Erlang/Efsm specification was representing caused some problems when generating test suites. The required execution time and memory made it impossible to create the largest test suite, All-Paths^{org}, on the available hardware with the implementation of the Erly Marsh tool, when not projecting the specification with a projection property. Although the use of a projection property was considered to only minimally influence quality of the generated test suites in this evaluation, it puts some light on a potential problem with coverage based model-based testing methods that require searching the complete model.

On Random testing: An interesting observation is that for smaller test suites of similar size the coverage based test suites performs better than both the event weighted and test case weighted random test suites. This is true, both with respect to source code coverage and bug fixes exposed. In smaller test suites each test case becomes relatively more important. To generate a good small test suite, it is an advantage to use some coverage criteria distributing test cases over the model. However, for larger test suites, event weighted randomized test suites perform as well as coverage based test suites with respect to bug fixes exposed and source code coverage. Typical for the event weighted randomized test suites were that they had a large number of shorter test cases, but only few longer test cases. Thus, it seems that many bug fixes are exposed in shorter test cases and bug fixes that only occur in long test cases are easy to find. 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.

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, see Figure 8.6. 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}). 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^{ew}_{100k} in total still exposes many different bug fixes can probably be explained by that there exists some "hard to find" short test cases.

Generating a random test suite can be expensive if there exists many complicated constraints on parameters, deciding whether a parameter combination is included in the model or not. In such cases many random selections of parameter values will be infeasible test cases, not covered by the model. Further, it may not be easy to decide if a test case is feasible or not. In our case, first randomly assigning values to all parameters then validate the selected parameter value combination to also correspond to a feasible test case turned out to be very expensive. Reason was that only 1 of approximately 8000 parameter completely random value combinations of the symbolic parameters 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 fair randomization. It can further be noted that also combinatorial testing techniques (see Section 1.5.3) suffer from similar problems.

On Manual testing: 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. 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 also expose rather few bug fixes (30%) which can be explained by the fact that these test cases had been previously executed.

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.

Due to the evaluation was limited to a single SUT one must be careful with empirical conclusions from the measured results. Still, the preliminary results where achieved on the type of larger industrial system, considered the target of this research. It is therefore our belief our results will hold in general.

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 to 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/Efsm 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 small specification and let it incrementally grow. Whenever it grow 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/Efsm. 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 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.

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`). It can be noted a property model over e.g., A-MLC would need to express all the details on state transitions, guards to input events etc. as the ERLANG/EFSM 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 specfication 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 testig 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.

9.2.1 Results

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 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 slighly. 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 where 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 cause 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 wich 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 11, Shafique 13]. 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 99a] 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 it 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 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 [de Vries 98, Tretmans 02], later rewritten into JTorX [Belinfante 10]. Both TGV and TorX are based on IOLTS (Input Output Labeled Transition System) and use the **ioco** 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 **ioco** 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 MurΦ 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] 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 [ETS 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 [de 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.

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.

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 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. A mutated implementation is also used the industrial case study presented in [Heimdahl 04]. A conclusion is that random testing may sometimes perform better than coverage based testing due to the structure of the specification and inadequate coverage criteria.

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 handcrafted 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 indi-

cate 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 achivements 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. Let us first summarize the results.

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 Section 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 also was advantageous in 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 on an EFSM, in Section 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 Section 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 Section 5. Here we defined symbolic test cases and symbolic test suites and gave a symbolic state space exploration algorithm for test suite generation. We also defined coverage when symbolically executing an EFSM with an observer. 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 observer states.

We have given an overview of Erly Marsh in Section 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, also integrating possibilities to validate the Erlang/Efsm specification, concretize and execute generated abstract test suites, and generate reports on the outcome of the executions on both individual test cases and test suites, The setup for this case study, where we use our techniques for model-based test suite generation, was presented in Section 7. The results of using Erly Marsh on this case study was presented in Section 8. In Section 9 we compared these results with other tools for testing 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 allow to make compact representations of specifications both of parameters in input events, and 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/Efsm we can use this to specify different input event types into different function clauses, to get more intuitive mappings to edges in an EFSM.

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/Efsm 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/Efsm 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 exists 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 Section 4. In fact, with the limitations implied by what can be expressed in Erlang/Efsm, 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. For other circumstances 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 where we can distinguish the use of

- *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., status codes from standards where we a clear understanding of the implementation, and we could use larger domains in the specification, without increasing 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 the symbolic representation offered by the NDD:s 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. Thereby limiting 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 used 9 abstraction macros to increase readability of the specification and limit the state space. We also 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 impossible for us to handle such a large number of abstract test cases with test suites generated off-line.

11.5 Concretization of Generated Test Cases

For any Erlang/Efsm specification representing an abstraction of a SUT there is a need to create concretization of any input to the SUT (and an abstraction of 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/Efsm 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 expressions 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

To efficiently execute and verify executed test cases and test suites requires good tool support. Particularly in an industrial setting it is important with necessary tool support covering all components involved in model-based testing, see Section 1.5. A considerable amount of time and effort has been spent on Erly Marsh to create a tool that, integrates test suite generation, concretization, execution and verification, after a model has been created. Thus, allowing for a high degree of automating testing.

When executing test cases concurrently there is a need to separate different test cases. In Erly Marsh this is solved when creating a concretiza-

tion by using parameters in concrete elements to carry a test identifier. 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

There exists a large number of different coverage criteria and other strategies for test suite generation. In the case study we applied a subset of these and made comparisons.

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 on faults, with a potential 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/Efsm. The relatively few faults found with the Dialyzer tool in Section 9.2 also indicate that 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 therefore not try to designate a coverage criteria "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. Our experience with random test suites from Section 8.1 showed that generally, random test suites required more test cases than partition based test suites to find the same number of bug fixes. For larger test suites we had

mixed results, depending on how the random test suites were generated. For larger test suites, randomly generating test suites with a technique favoring smaller test cases performed equally well as partition based. On the other hand, randomly generating test suites with a technique favoring longer test cases performed much worse. 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 specification to verificating 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 contribute in 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, see Figure 11.1. Nested states can then be specified in separate modules. There exists other examples of possible extensions of Erlang/Efsm. 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

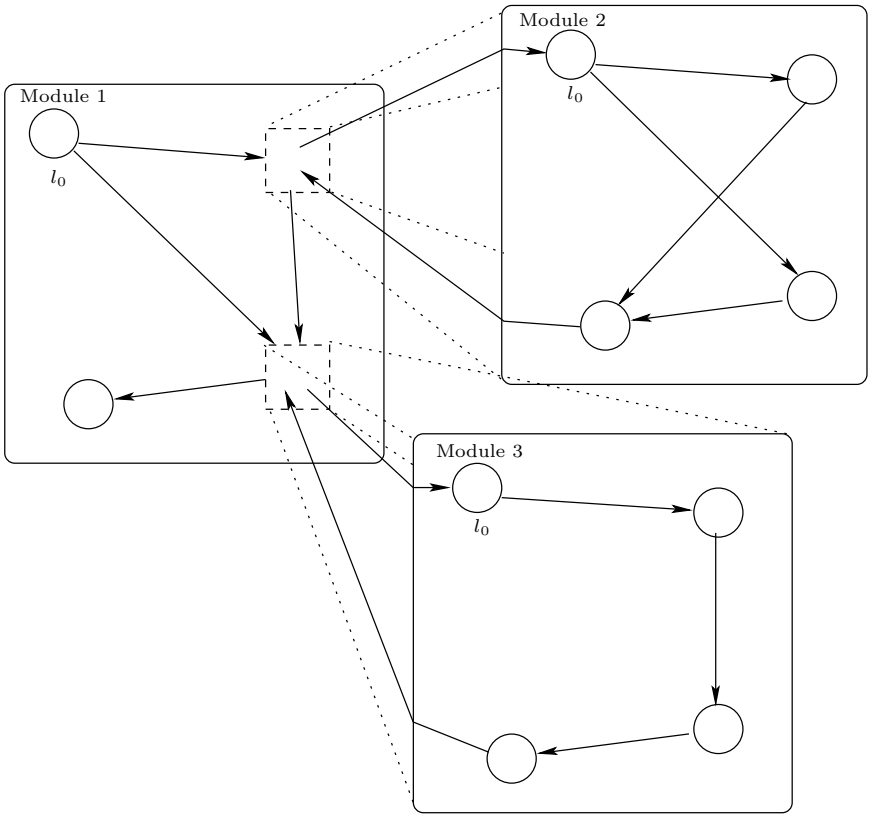


Figure 11.1. Three modules (compare with ERLANG modules), each defining a different state machine, outlining how ERLANG/EFSM possibly can be extended to support hierarchically state machines.

explore some of the large number of efficient representations of different subsets of expressions found in the literature.

Creation of an ERLANG/EFSM specification

A major cost involved with model-based testing is the (manual) creation of a specification. To consider tool assistance when creating a specification may therefore be an attractive alternative.

There exists several approaches for automatically generating a formal model from an existing implementation. For example, *predicate abstraction* is used by Bandera [Corbett 00] to extract a formal model from Java source code and SLAM [Ball 05, Ball 01] to extract a formal model from C/C# source code. Another approach is to use *regular inference*, [Bohlin 09], to create a finite state machine without access to the source code. In a similar fashion it would be possible to build a tool that automatically generates an ERLANG/EFSM specification from an existing SUT.

In Section 2.11 we showed that it was straightforward to create an Erlang/Otp module implementing the `gen_fsm` behavior from an Erlang/Efsm specification. It would also be possible to derive an Erlang/Efsm model from an Erlang program. In the simplest case, we can translate the Erlang program by reversing the rules in Section 2.11. But if the Erlang program utilize features not part of Erlang/Efsm, or if we wish to reduce the state space, an abstraction is needed. Assuming and Erlang program with a single Erlang process a possibility would be to use the abstraction macros in Section 2.3.3. More advanced modeling would require a dedicated tool.

An advantage with an automatically created model in Erlang/Efsm would be that the test suite generation presented in this thesis could be used as is. However, as also discussed in Section 1.5.1, the set of failures typically detected by a test suite from an automatically generated specification would be a subset of the failures detected by a test suite from a specification whose creation has been supervised by an expert. Mainly we would detect failures in the implementation such as program crashes, assertion violations, and non-termination. That is, the scope of the testing moves from focusing on finding faults only on a requirement/functional level, to lower level faults, similar to those that can be found by static analysis tools (e.g., Dialyzer).

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 support

a set of hard coded observer predicates. We did also not yet include a constraint solver simply because it was not needed for the A-MLC Erlang/Efsm specification in the evaluation. A consequence is that Erly Marsh currently can not generate all test suites that is possible to express with Erlang/Efsm.

Below some further ideas on how observers could be used to generate other types of test suites is further briefly outlined.

Regression testing

Regression testing a system, i.e., re-test a selected subset of test cases, after changes have been made, is an important form of testing. Regression testing of a system can be motivated whenever functionality is changed, by updating the specification, or when a fault is corrected in the SUT and we do not wish to re-run a complete test suite. Often corrected faults also tends to reappear over time, which motivates an existing test suite to be extended over time. Regression testing is particular important for larger systems. For any regression test technique to scale it therefore makes sense to use a high-level representation of the SUT see e.g., [Orso 04]. In model-based testing, the model can be used to find out which existing test cases that are affected by an update of the specification. A brute-force approach would be to include all possibly affected test cases. Assume TS is an old existing test suite and TS' is a test suite created after some changes have been made to the specification. We can then define a regression test suite $RegTS$ as

$$RegTS = TS' - (TS \cap TS')$$

This can be fine tuned, because all test cases in $RegTS$ may not have any dependencies on changes made, see e.g., [Chen 07]. To find $RegTS$, or a more fine-tuned version, we could then introduce some means to identify what has changed in the updated specification. It would then be possible to define observer automata that specifically monitor those parts of the specification that are affected by the update. See also e.g., [Rothermel 04] for a case study between a number of different regression testing strategies.

Test suite detecting unwanted Feature Interactions

Complex software systems, such as a telephony systems, are preferably divided into an architecture where a number of features can be identified. Potentially these features also interact with each other. The feature interaction problem can be expressed as that of managing such complex systems, where new features are continuously added over time. Formally model a SUT and expressing a formal definition of feature interaction as a coverage criteria would generate a test suite with test cases where

feature interaction occurs in the model. The existence of such test suite seems beneficial when resolving found feature interactions.

In an earlier work [Blom 95] we formally defined feature interaction in a non-deterministic model of a system. In this thesis, however, we have assumed a deterministic model. Thus, in order to utilize these feature interaction definitions we would need to extend the use of ERLANG/EFSM and test suite generation algorithm to non-deterministic models. We would further need to assure we formally can express properties necessary for feature interaction as observers.

Other approaches to express feature interaction as coverage criteria in model-based testing exists, see e.g., [Lochau 10].

An observer automata algebra

It may be convenient to allow multiple observers to be used concurrently when exploring the state space of an EFSM. With multiple observers we will generate a set of test cases for each observer. Further, one can assume that we may be interested in combinations of both the union and intersection of all these sets of tests cases. This leads to the possibility to allow control of test suite generation by an *observer algebra*.

We will here just outline how such an observer algebra could be defined. First, recall from Section 5.5.1 that the observer automata could be used as a filter. We could then

- let the elements of the algebra be observer automata where we e.g., can distinguish (1) observers O, that adds test cases that contributes by cover some, previously uncovered, coverage item to a test suite, and (2) filters F, that restricts a test suite by requiring all test cases part of the test suite to fulfill properties expressed by the observer automata, and
- let the operations $+$ be the union of two observer automatas and the operation $*$ be the intersection of two observer automatas.

For example, assume two observers O_1, O_2 and two filters F_1, F_2. Possible combinations of these observer automatas for test suite generation then includes:

- $F_1 * (O_1 + O_2)$ that is the union of all test cases generated by the observers O_1, O_2, that also passes filter F_1
- $(F_1 * O_1) + (F_2 * O_2)$ that is the union of all test cases generated by observer O_1 filtered by F_1, and observer O_2 filtered by F_2
- $F_1 * F_2$ that are all the resulting test cases from filtering with both F_1 and F_2.
- Note that $F_1 + O_1$ are all the resulting test cases from first filtering with F_1 and then generating with O_1. This is different from $F_1 * O_1$ where the filter F_1 is applied on the set generated by O_1.

Testing with different granularity
In Section 2.10 we showed how edges in the EFSM were derived and represented as edge clauses. That is, a single computation step is the complete execution triggered by an input event to the tuple `{next_state,l'}` representing a transition to the next location l'. However, this execution defining a single computation step is not the only possible choice. Consider, for example, if we additionally defined each `case`- or `if` expression to represent a decision point, where execution *to* the decision point represent one computation step and execution *from* the decision point is another computation step.

Example 11.1 Consider a transition clause from an Erlang/Efsm specification:

```
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.
```

With the definition of an edge clause used elswhere in this thesis we would end up with the following two edge clauses:

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

With a definition of an edge clause where `case`- or `if` expression represent a decision point we instead get the following three edge clauses:

```
morning(wakeup,TDay) ->
    Progress=0, Stamina=2, Day=TDay,{next_state,morning_int1}.

morning_int1() when daytype(Day)==collect ->
    checkout(Day), {next_state,workUU};
morning_int1() when daytype(Day)==leave ->
    {next_state,preschool}.
```

where `morning_int1` is an additional "internal" location. A transition to this internal location occurs on an internal "empty" event, not visible

outside the SUT.

□

The introduction of invisble internal events between visible input events leads to a higher "granularity" of the execution represented by a computation step. Thus, for transition clauses with multiple `case`- or `if` expressions we would further be able to validate execution of the specification. Considering a finer granularity level would also lead to a closer relationship with some existing overage criteria on guards, see Section 4.1.1. To allow for observers to monitors internal events the superposition between edge clauses and observer edges also needs to be redefined.

Increasing the level of granularity may be expected to increase the size of generate test suites for most coverage criteria. In order to verify execution of the internal events in a test suite a SUT must additionally be instrumented with some kind of probes, sending data to the test environment.

Optimizing test suite generation

The problem of generating a test suite with a minimum cost for a coverage criterion covering a (fixed) set of coverage items can be formulated as a variant of the *minimum spanning tree* problem (see e.g., [Cormen 01]). Given a connected, undirected graph, a spanning tree of that graph is a sub graph which is a tree and connects all the vertices's together. A test suite is a spanning tree over a set of coverage items because all test cases start in the same initial state and end in a stop state (a leaf in the tree).

By assigning a cost to each computation step, i.e., a number representing how favorable it is we can compute the cost of a spanning tree by computing the sum of the costs of the edges. Given a test suite TS with test cases $t_1, \ldots, t_{i-1}$, the costs of TS can be calculated by a function

$$\sum_{t_i \in TS} cost(t_i)$$

where a function *cost* to calculate the cost on which each test case contributes to the cost of a test suite, e.g., number of output events. A minimum spanning tree spanning tree is then a spanning tree with a cost less than or equal to the cost of every other spanning tree.

References

[3GP 99] 3GPP. *Mobile Application Part (MAP) specification*, April 1999.

[3GP 00] 3GPP. *Functional stage 2 description of Location Services (LCS)*, July 2000.

[Aichernig 09] BernhardK. Aichernig and He Jifeng. *Mutation testing in UTP. Formal Aspects of Computing*, vol. 21, no. 1-2, pages 33–64, 2009.

[Aichernig 12] Bernhard K. Aichernig. *Model-Based Mutation Testing: Theory and Application.* Habilitation thesis, Graz University of Technology, Austria, January 2012.

[Ammann 03] Paul Ammann, Jeff Offutt, and Hong Huang. *Coverage Criteria for Logical Expressions.* In *Proceedings of the 14th International Symposium on Software Reliability Engineering*, ISSRE '03, pages 99–, Washington, DC, USA, 2003. IEEE Computer Society.

[Ammann 08] Paul Ammann and Jeff Offutt. *Introduction to Software Testing.* Cambridge University Press, New York, NY, USA, 1 edition, 2008.

[Ammons 03] Glenn Ammons, David Mandelin, Rastislav Bodík, and James R. Larus. *Debugging temporal specifications with concept analysis. ACM SIGPLAN Notices*, vol. 38, no. 5, pages 182–195, 2003.

[Anand 13] Saswat Anand, Edmund K Burke, Tsong Yueh Chen, John Clark, Myra B Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, Phil McMinn, et al. *An orchestrated survey of methodologies for automated software test case generation. Journal of Systems and Software*, vol. 86, no. 8, pages 1978–2001, 2013.

[Arts 06] T. Arts, J. Hughes, J. Johansson, and U. Wiger. *Testing Telecoms Software with Quviq QuickCheck.* In *Proc. 2006 ACM SIGPLAN workshop on Erlang*, Portland, Oregon, USA, Sep. 2006.

[Arvaniti 11] Eirini Arvaniti. Automated Random Model-Based Testing of Stateful Systems. Diploma thesis, National Technical University of Athens, School of Electrical and Computer Engineering, July 2011. `http://artemis.cslab.ntua.gr/el_thesis/artemis.ntua.ece/DT2011-0142/DT2011-0142.pdf`.

[Ball 01] Thomas Ball, Rupak Majumdar, Todd Millstein, and Sriram K. Rajamani. *Automatic Predicate Abstraction of C Programs.* In *PLDI 2001*, pages 203–213, 2001.

[Ball 05] T. Ball, O. Kupferman, and G. Yorsh. *Abstraction for Falsification.* Technical Report MSR-TR-2005-50, Microsoft, 2005.

[Barklund 99] J. Barklund and R. Virding. ERLANG 4.7.3 reference manual. Draft (0.7), Ericsson Computer Science Laboratory, 1999.

[Beizer 90] B. Beizer. *Software Testing Techniques.* Van Nostrand Reinhold, New York, 1990.

[Belinfante 10] Axel Belinfante. *JTorX: A Tool for On-line Model-driven Test Derivation and Execution.* In *Proceedings of the 16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'10, pages 266–270, Berlin, Heidelberg, 2010. Springer-Verlag.

[Bertolino 07] Antonia Bertolino. *Software Testing Research: Achievements, Challenges, Dreams.* In *FOSE '07: 2007 Future of Software Engineering*, pages 85–103, Washington, DC, USA, 2007. IEEE Computer Society Press.

[Beyer 04] Dirk Beyer, Adam J. Chlipala, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. *The Blast query language for software verification.* In *LNCS*, pages 2–18. Springer, 2004.

[Beyer 13] Dirk Beyer, Andreas Holzer, Michael Tautschnig, and Helmut Veith. *Information Reuse for Multi-goal Reachability Analyses.* In Matthias Felleisen and Philippa Gardner, editors, *Programming Languages and Systems*, volume 7792 of *Lecture Notes in Computer Science*, pages 472–491. Springer Berlin Heidelberg, 2013.

[Binder 99] R. Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools.* Addison Wesley, 1999.

[Bishop 05] Steve Bishop, Matthew Fairbairn, Michael Norrish, Peter Sewell, Michael Smith, and Keith Wansbrough. *Rigorous specification and conformance testing techniques for network protocols, as applied to TCP, UDP, and sockets. SIGCOMM Comput. Commun. Rev.*, vol. 35, no. 4, pages 265–276, 2005.

[Blom 94] J. Blom, B. Jonsson, and L. Kempe. *Using Temporal Logic for Modular Specification of Telephone Services.* In L.G. Bouma and H. Velthuijsen, editors, *Feature Interactions in Telecommunications Systems.* IOS Press, Amsterdam, Netherlands, May 1994.

[Blom 95] J. Blom, R. Bol, and L. Kempe. *Automatic Detection of Feature Interactions in Temporal Logic.* In K.E. Cheng and T. Ohta, editors, *Feature Interactions in Telecommunications Systems III.* IOS Press, Amsterdam, Netherlands, October 1995.

[Blom 96a] J. Blom, R. Bol, B. Jonsson, and J. Nyström. *Creation of Dependent Features.* In *Proceedings of RVK '96,*

RadioVetenskap och Kommunikation'96, Luleå, Sweden, June 1996.

[Blom 96b] J. Blom and B. Jonsson. *Constraint Oriented Temporal Logic Specification.* In M. Broy, S. Merz, and K. Spies, editors, *Formal Systems Specification, The RPC-Memory Specification Case Study*, volume 1169 of *Lecture Notes in Computer Science*, pages 161–182. Springer-Verlag, 1996.

[Blom 97] J. Blom. *Formalisation of Requirements with Emphasis on Feature Interaction Detection.* In P. Dini, R. Boutaba, and L. Logrippo, editors, *Feature Interactions in Telecommunications Systems IV*, pages 61–77. IOS Press, Amsterdam, Netherlands, 1997.

[Blom 03] J. Blom and B. Jonsson. *Automated Test Generation for Industrial* ERLANG *Applications.* In *Proc. 2003 ACM SIGPLAN workshop on Erlang*, pages 8–14, Uppsala, Sweden, Aug. 2003.

[Blom 04] J. Blom, A. Hessel, B. Jonsson, and P. 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.

[Blom 16] J. Blom, B. Jonsson, and S.-O. 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.

[Boberg 08] Jonas Boberg. *Early fault detection with model-based testing.* In *Proceedings of the 7th ACM SIGPLAN workshop on Erlang*, ERLANG '08, pages 9–20, New York, NY, USA, 2008. ACM.

[Bohlin 09] Therese Bohlin. *Regular Inference for Communication Protocol Entities.* PhD thesis, Dept. of Computer Systems, Uppsala University, Sweden, Uppsala, Sweden, 2009.

[Bolognesi 89] T. Bolognesi and E. Brinksma. *Introduction to the ISO Specification Language LOTOS. Computer Networks*, vol. 14, no. 1, Jan. 1989.

[Bouquet 03] F. Bouquet and B. Legeard. *Reification of Executable Test Scripts in Formal Specification-Based Test Generation: The Java Card Transaction Mechanism Case Study.* In *FME 2003*, volume 2805 of *Lecture Notes in Computer Science*, pages 778–795. Springer-Verlag, 2003.

[Boyapati 02] Rasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. *Korat: Automated testing based on Java predicates.* In *Proc. International Symposium on Software Testing and Analysis (ISSTA*, pages 123–133. ACM Press, 2002.

[Briand 07] Lionel C. Briand. *A Critical Analysis of Empirical Research*

in Software Testing. In *First International Symposium on Empirical Software Engineering and Measurement*, pages 376–387. IEEE Computer Society Press, September 2007.

[Broy 04] M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, and A. Pretschner, editors. *Model-Based Testing of Reactive Systems*, volume 3472 of *Lecture Notes in Computer Science*. Springer-Verlag, 2004.

[Brucker 13] Achim D. Brucker and Burkhart Wolff. *On theorem prover-based testing. Formal Aspects of Computing*, vol. 25, no. 5, pages 683–721, 2013.

[Bryant 86] R.E. Bryant. *Graph-Based Algorithms for Boolean Function Manipulation. IEEE Trans. on Computers*, vol. C-35, no. 8, pages 677–691, Aug. 1986.

[Burch 92] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L. J. Hwang. *Symbolic Model Checking:* 10^{20} *States and Beyond. Information and Computation*, vol. 98, no. 2, pages 142–170, June 1992.

[Chen 07] Yanping Chen, Robert L. Probert, and Hasan Ural. *Model-based regression test suite generation using dependence analysis*. In *A-MOST '07: Proceedings of the 3rd international workshop on Advances in Model-Based Testing*, pages 54–62, New York, NY, USA, 2007. ACM.

[Chilenski 94] John Joseph Chilenski and Steven P. Miller. *Applicability of Modified Condition/Decision coverage to software testing. Software Engineering Journal*, vol. 9, no. 5, pages 193–200, September 1994.

[Ciupa 07] Ilinca Ciupa, Andreas Leitner, Manuel Oriol, and Bertrand Meyer. *Experimental assessment of random testing for object-oriented software*. In *ISSTA '07: Proceedings of the 2007 international symposium on Software testing and analysis*, pages 84–94, New York, NY, USA, 2007. ACM.

[Clarke 89] Lori A. Clarke, Andy Podgurski, Debra J. Richardsson, and Steven J. Zeil. *A Formal Evaluation of Data Flow Path Detection Criteria. IEEE Trans. on Software Engineering*, vol. SE-15, no. 11, pages 1318–1332, Nov 1989.

[Clarke 99] E.M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, Dec. 1999.

[Clarke 02] D. Clarke, T. Jéron, V. Rusu, and E. Zinovieva. *STG: A Symbolic Test Generation Tool*. In *Proc. TACAS '02,* 8^{th} *Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, volume 2280 of *Lecture Notes in Computer Science*, pages 470–475. Springer-Verlag, 2002.

[Cohen 97] D.M. Cohen, S.R. Dalal, M.L. Fredman, and G.C. Patton. *The AETG system: An approach to testing based on combinatorial design. IEEE Trans. on Software Engineering*, vol. SE-23, no. 7, pages 437–444, July 1997.

[Consel 93] C. Consel and O. Danvy. *Tutorial Notes on Partial*

Evaluation. In *Proc.* 20^{th} *ACM Symp. on Principles of Programming Languages*, 1993.

[Corbett 00] J.C. Corbett, M.B. Dwyer, J. Hatcliff, S. Laubach, C.S. Pasareanu, Robby, and H. Zheng. *Bandera : Extracting Finite-state Models from Java Source Code*. In *Proc. 22nd Int. Conf. on Software Engineering*, June 2000.

[Cormen 01] Thomas Cormen, Charles Leiserson, Ronald Rivest, and Clifford Stein. *Introduction to Algorithms*. MITP, second edition, 2001.

[Cronqvist 04] Mats Cronqvist. *Troubleshooting a large* ERLANG *system*. In *Proceedings of the 2004 ACM SIGPLAN workshop on Erlang*, ERLANG '04, pages 11–15, New York, NY, USA, 2004. ACM.

[de Alfaro 01] Luca de Alfaro and Thomas A. Henzinger. *Interface automata. SIGSOFT Softw. Eng. Notes*, vol. 26, no. 5, pages 109–120, 2001.

[de Vries 98] R.G. de Vries and Jan Tretmans. *On-the-Fly Conformance Testing Using SPIN*. In *SPIN Model Checking and Software Verification: Proc.* 4^{th} *Int. SPIN Workshop*, 1998.

[Dijkstra 75] Edsger W. Dijkstra. *Guarded Commands, Nondeterminacy and Formal Derivation of Programs. Commun. ACM*, vol. 18, no. 8, pages 453–457, August 1975.

[D'Souza 03] Deepak D'Souza and Madhavan Mukund. *Checking Consistency of SDL+MSC Specifications*. In Thomas Ball and Sriram K. Rajamani, editors, *SPIN*, volume 2648 of *Lecture Notes in Computer Science*, pages 151–165. Springer, 2003.

[E. Asarin 97] E. Asarin, M. Bozga, A. Kerbrat, O. Maler, M. Pnueli, and A. Rasse. *Data Structures for the Verification of Timed Automata*. In O. Maler, editor, *Hybrid and Real-Time Systems*, pages 346–360, Grenoble, France, 1997. Springer Verlag, LNCS 1201.

[El-Far 02] Ibrahim K. El-Far and J.A. Whittaker. *Model-based software testing. Encyclopedia of Software Engineering*, vol. 1, pages 825–837, 2002.

[Eri 15] Ericsson. *Erlang Open Telecom Platform*, October 2015.

[ETS 03] ETSI. *The Testing and Test Control Notation version 3*, 2003.

[Fernandez 96] J.-C. Fernandez, C. Jard, T. Jéron, and C. Viho. *Using On-the-fly Verification Techniques for the Generation of Test Suites*. In R. Alur and T. Henzinger, editors, *Computer Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, New Brunswick, New Jersey, USA, Jul./Aug. 1996. Springer-Verlag.

[Fernandez 97] J.-C. Fernandez, C. Jard, T. Jéron, and C. Viho. *An Experiment in Automatic Generation of Test Suites for Protocols with Verification Technology. Science of Computer*

Programming, vol. 29, 1997.

[Frankl 88] P.G. Frankl and E. J. Weyuker. *An Applicable Family of Data Flow Testing Criteria. IEEE Trans. on Software Engineering*, vol. 14, pages 1483–1498, October 1988.

[Frantzen 05] L. Frantzen, J. Tretmans, and T. Willemse. *Test Generation Based on Symbolic Specifications.* In *FATES 2004*, volume 3395 of *Lecture Notes in Computer Science*, pages 1–15. Springer-Verlag, 2005.

[Fraser 09] Gordon Fraser, Franz Wotawa, and Paul E. Ammann. *Testing with Model Checkers: A Survey. Softw. Test. Verif. Reliab.*, vol. 19, no. 3, pages 215–261, September 2009.

[Fredlund 01] Lars Åke Fredlund. *A Framework for Reasoning about* ERLANG *code.* PhD thesis, Royal Institute of Technology, Stockholm, Aug 2001.

[Friedman 02] G. Friedman, A. Hartman, K. Nagin, and T. Shiran. *Projected State Machine Coverage for Software Testing.* In *Proc. ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 134–143, 2002.

[Gargantini 01] A. Gargantini and E. Riccobene. *ASM-Based Testing: Coverage Criteria and Automatic Test Sequence. Journal of Universal Computer Science*, vol. 7, no. 11, pages 1050–1067, 2001.

[Gargantini 03] A. Gargantini, E. Riccobene, and S. Rinzivillo. *Using Spin to Generate Tests from ASM Specifications.* In E. Börger, A. Gargantini, and E. Riccobene, editors, *Proceedings of the 10th International Workshop on Abstract State Machines (ASM 2003)*, volume 2589 of *Lecture Notes in Computer Science*, pages 263–277. Springer-Verlag, 2003.

[George 12] Devaraj George. *On the effectiveness of specification-based structural test-coverage criteria as test-data generators for safety-critical systems.* PhD thesis, University of Minnesota, Dec 2012.

[Gnesi 02] Stefania Gnesi, Diego Latella, and Mieke Massink. *Modular semantics for a UML statechart diagrams kernel and its extension to multicharts and branching time model-checking. Journal of Logic and Algebraic Programming*, vol. 51, no. 1, pages 43–75, 2002.

[Goga 01] N. Goga. *Comparing TorX, Autolink, TGV and UIO Test Algorithms.* In *SDL '01: Proceedings of the 10th International SDL Forum Copenhagen on Meeting UML*, pages 379–402, London, UK, 2001. Springer-Verlag.

[Gotlieb 98] Arnaud Gotlieb, Bernard Botella, and Michel Rueher. *Automatic test data generation using constraint solving techniques. SIGSOFT Softw. Eng. Notes*, vol. 23, no. 2, pages 53–62, 1998.

[Grabowski 95] J. Grabowski, D. Hogrefe, I. Nussbaumer, and A. Spichiger. *Test Case Specification Based on MSCs and ASN.1.* In

R. Bræk and A. Sarma, editors, *SDL'95 with MSC in CASE, Proc. 7th SDL Forum*, pages 307–322, Oslo, Norway, Sept. 1995. Elsevier.

[Grieskamp 11] Wolfgang Grieskamp, Nicolas Kicillof, Keith Stobie, and Victor Braberman. *Model-based Quality Assurance of Protocol Documentation: Tools and Methodology. Softw. Test. Verif. Reliab.*, vol. 21, no. 1, pages 55–71, March 2011.

[Grindal 07] Mats Grindal. *Handling Combinatorial Explosion in Software Testing.* PhD thesis, Link ping University, Link ping, Sweden, 2007.

[Havelund 02] K. Havelund and G. Rosu. *Synthesizing Monitors for Safety Properties.* In J.-P. Katoen and P. Stevens, editors, *Proc. TACAS '02,* 8^{th} *Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, volume 2280 of *Lecture Notes in Computer Science*, pages 324–356. Springer-Verlag, 2002.

[Heimdahl 04] Mats P.E. Heimdahl, Devaraj George, and Robert Weber. *Specification Test Coverage Adequacy Criteria=Specification Test Generation Inadequacy Criteria?* In *Proc. Eighth IEEE International Symposium on High Assurance Systems Engineering (HASE '04)*, 2004.

[Herman 76] P.M. Herman. *A data flow analysis approach to program testing. Australian Computer J.*, vol. 8, no. 3, November 1976.

[Hessel 07] A. Hessel. *Model-Based Test Case Generation for Real-Time Systems.* PhD thesis, Dept. of Computer Systems, Uppsala University, Sweden, Uppsala, Sweden, 2007.

[Holzmann 97] G.J. Holzmann. *The Model Checker SPIN. IEEE Trans. on Software Engineering*, vol. SE-23, no. 5, pages 279–295, May 1997.

[Holzmann 02] G.J. Holzmann and H. Smith. *An Automated Verification Method for Distributed Systems Software Based on Model Extraction. IEEE Trans. on Software Engineering*, vol. 28, pages 1–14, 2002.

[Holzmann 03] G.J. Holzmann. *The SPIN Model Checker.* Addison-Wesley, September 2003.

[Hong 02] H.S. Hong, I. Lee, O. Sokolsky, and H. Ural. *A Temporal Logic Based Theory of Test Coverage and Generation.* In J.-P. Katoen and P. Stevens, editors, *Proc. TACAS '02,* 8^{th} *Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, volume 2280 of *Lecture Notes in Computer Science*, pages 327–341, London, UK, 2002. Springer-Verlag.

[Hong 03] H.S. Hong, S.D. Cha, I. Lee, O. Sokolsky, and H. Ural. *Data Flow Testing as Model Checking.* In *ICSE'03: 25th Int. Conf. on Software Enginering*, pages 232–242, May 2003.

[Hughes 10] John Hughes. *Software testing with QuickCheck.* In *Central*

European Functional Programming School, pages 183–223. Springer, 2010.

[Huima 07] Antti Huima. *Implementing Conformiq Qtronic*. In Alexandre Petrenko, Margus Veanes, Jan Tretmans, and Wolfgang Grieskamp, editors, *Testing of Software and Communicating Systems*, volume 4581 of *Lecture Notes in Computer Science*, pages 1–12. Springer Berlin / Heidelberg, 2007. 10.1007/978-3-540-73066-8_1.

[IBM 11] IBM. *Rational Tau*, 2011. http://www.ibm.com/software/products/en/ratitau.

[Ip 99] C. Norris Ip and David L. Dill. *Verifying Systems with Replicated Components in Murφ. Formal Methods in System Design*, vol. 14, no. 3, May 1999.

[ITU-T 99a] ITU-T. *Recommendation Z.100, Specification and Description Language (SDL)*, November 1999.

[ITU-T 99b] ITU-T. *Recommendation Z.120, Message Sequence Charts (MSC)*, Nov. 1999.

[Jacky 08] Jonathan Jacky, Margus Veanes, Colin Campbell, and Wolfram Schulte. *Model-Based Software Testing and Analysis with C#*. Cambridge University Press, Jan. 2008.

[Janhunen 11] Tomi Janhunen, Ilkka Niemelä, Johannes Oetsch, Jörg Pührer, and Hans Tompits. *Random vs. Structure-Based Testing of Answer-Set Programs: An Experimental Comparison*. In James Delgrande and Wolfgang Faber, editors, *Logic Programming and Nonmonotonic Reasoning*, volume 6645 of *Lecture Notes in Computer Science*, pages 242–247. Springer Berlin / Heidelberg, 2011. 10.1007/978-3-642-20895-9_26.

[Jantsch 98] A. Jantsch, S. Kumar, I. Sander, B. Svantesson, J. Oberg, A. Hemani, P. Ellervee, and M. O'Nils. *Comparison of six languages for system level descriptions of telecom systems*. In *First International Forum on Design Languages*, volume 2, pages 139–148, Lausanne, Switzerland, September 1998.

[Jantsch 00] Axel Jantsch and Ingo Sander. *On the Roles of Functions and Objects in System Specification*. In *in Proceedings of the International Workshop on Hardware/Software Codesign*, 2000.

[Jard 05] C. Jard and T. Jeron. *TGV: theory, principles and algorithms. Int. Journal on Software Tools for Technology Transfer*, vol. 7, no. 4, pages 297–315, 2005.

[Kadono 09] Masaya Kadono, Tatsuhiro Tsuchiya, and Tohru Kikuno. *Using the NuSMV Model Checker for Test Generation from Statecharts*. In *Proceedings of the 2009 15th IEEE Pacific Rim International Symposium on Dependable Computing*, PRDC '09, pages 37–42, Washington, DC, USA, 2009. IEEE Computer Society.

[King 76] James C. King. *Symbolic Execution and Program Testing. Communications of the ACM*, vol. 19, no. 7, pages 385–394, July 1976.

[Knoop 96] J. Knoop, B. Steffen, and J. Vollmer. *Parallelism for Free: Efficient and Optimal Bitvector Analyses for Parallel Programs. ACM Transactions on Programming Languages and Systems*, vol. 18, no. 3, pages 268–299, 1996.

[Koch 98] Beat Koch, Jens Grabowski, Dieter Hogrefe, and Michael Schmitt. *Autolink - A Tool for Automatic Test Generation from SDL Specifications.* In *Proceedings of 'Workshop on Industrial Strength Formal Specication Techniques (WIFT'98)', October 21-23, Boca*, pages 21–23, 1998.

[Lai 02] R. Lai. *A survey of communication protocol testing. Journal of Systems and Software*, vol. 62, pages 21–46, 2002.

[Larsen 97] K.G. Larsen, P. Pettersson, and W. Yi. *UPPAAL in a Nutshell. Software Tools for Technology Transfer*, vol. 1, no. 1-2, 1997.

[Laski 83] J. W. Laski and B. Korel. *A Data Flow Oriented Program Testing Strategy. IEEE Trans. on Software Engineering*, vol. SE-9, no. 3, pages 347–354, May 1983.

[Leavens 99] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. *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, 1999.

[Lin 96] Huimin Lin. *Symbolic transition graph with assignment.* In *Proceedings of CONCUR'96, LNCS 1119*, pages 50–65. Springer-Verlag, 1996.

[Lindahl 06] Tobias Lindahl and Konstantinos Sagonas. *Practical type inference based on success typings.* In *Proceedings of the 8th ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming*, pages 167–178, New York, NY, USA, 2006. ACM Press.

[Lochau 10] Malte Lochau and Ursula Goltz. *Feature Interaction Aware Test Case Generation for Embedded Control Systems. Electron. Notes Theor. Comput. Sci.*, vol. 264, no. 3, pages 37–52, December 2010.

[Mäkinen 07] M. A. Mäkinen. Model based approach to software testing. Master's thesis, Helsinki University of Technology, May 2007.

[Malaiya 02] Yashwant K. Malaiya, Michael Naixin Li, James M. Bieman, and Rick Karcich. *Software Reliability Growth with Test Coverage. IEEE Transactions on Reliability*, vol. 51, pages 420–426, 2002.

[Mandl 85] O. Mandl. *Orthogonal Latin Squares: An experiment design to compiler testing. Communications of the ACM*, vol. 28, no. 10, pages 1054–1058, Oct. 1985.

[Mandrioli 95] D. Mandrioli, S. Morasca, and A. Morzenti. *Generating Test Cases for Real-Time Systems from Logic Specifications. ACM Trans. on Computer Systems*, vol. 13, no. 4, pages 365–398, Nov. 1995.

[Marinescu 15] Raluca Marinescu, Cristina Seceleanu, H l ne Le Guen, and Paul Pettersson. *A Research Overview of Tool-Supported Model-based Testing of Requirements-based Designs. Advances in Computers*, vol. 98, pages 89 – 140, 2015.

[Mauw 97] S. Mauw and M.A. Reniers. *Operational Semantics for MSC'96.* In A. Cavalli and D. Vincent, editors, *SDL'97 - Time for Testing - SDL, MSC and Trends*, pages 135–152. Elsevier, Sept. 1997.

[Myers 79] Glenford J. Myers. *Art of Software Testing.* John Wiley & Sons, Inc., New York, NY, USA, 1979. First edition, 3rd edition from 2011.

[Ntafos 88] S. C. Ntafos. *A Comparison of Some Structural Testing Strategies. IEEE Trans. Softw. Eng.*, vol. 14, no. 6, pages 868–874, 1988.

[Ntafos 01] Simeon C. Ntafos. *On Comparisons of Random, Partition, and Proportional Partition Testing. IEEE Trans. Softw. Eng.*, vol. 27, no. 10, pages 949–960, 2001.

[OMA 04] OMA. *Mobile Location Protocol*, April 2004. Version 1.1.

[OMG 03] OMG. *Unified Modeling Language, UML*, Mar 2003.

[Orso 04] Alessandro Orso, Nanjuan Shi, and Mary Jean Harrold. *Scaling regression testing to large software systems. SIGSOFT Softw. Eng. Notes*, vol. 29, no. 6, pages 241–251, 2004.

[Plotkin 81] G. Plotkin. *A Structural Approach to Operational Semantics.* Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, Denmark, 1981.

[Prenninger 05] W. Prenninger and A. Pretschner. *Abstractions for Model-Based Testing. Electronic Notes in Theoretical Computer Science*, vol. 116, pages 59–71, Jan 2005. Proceedings of the International Workshop on Test and Analysis of Component Based Systems (TACoS 2004).

[Pretschner 01] A. Pretschner. *Classical search strategies for test case generation with constraint logic programming.* In *Proc. Formal Approaches to Testing of Software, FATES '01*, pages 47–60, Aalborg, Denmark, August 2001.

[Pretschner 05] A. Pretschner, W. Prenninger, S. Wagner, C. Kühnel, M. Baumgartner, B. Sostawa, R. Zölch, and T. Stauner. *One evaluation of model-based testing and its automation.* In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 392–401, New York, NY, USA, 2005. ACM.

[Rapin 03] N. Rapin, C. Gaston, A. Lapitre, and J.-P. Gallois. *Behavioural Unfolding of Formal Specifications Based on*

Communicating extended automata. In *Proceedings of the 1st International Workshop on Automated Technology for Verification and Analysis*, Tapei, Taiwan, Dec. 2003.

[Rapps 85] Sandra Rapps and Elaine J. Weyuker. *Selecting software test data using data flow information. IEEE Trans. on Software Engineering*, vol. 11, no. 4, pages 367–375, April 1985.

[Rayadurgam 01] S. Rayadurgam and M. P. Heimdahl. *Test-Sequence Generation from Formal Requirements Models*. In *Proceedings of the 6th IEEE International Symposium on High-Assurance Systems Engineering (HASE 2001)*, pages 23–31. IEEE Computer Society Press, 2001.

[Richardson 92] D.J. Richardson, S. Leif-Aha, and T.O. O'Malley. *Specification-Based Test Oracles for Reactive Systems*. In *Proc. 14th Int. Conf. on Software Engineering*, pages 105–118. ACM Press, May 1992.

[Rothermel 04] Gregg Rothermel, Sebastian Elbaum, Alexey G. Malishevsky, Praveen Kallakuri, and Xuemei Qiu. *On test suite composition and cost-effective regression testing. ACM Trans. Softw. Eng. Methodol.*, vol. 13, no. 3, pages 277–331, 2004.

[Rusu 00] V. Rusu, L. du Bousquet, and T. Jéron. *An Approach to Symbolic Test Generation*. In *Int. Conf. on Integrating Formal Methods*, volume 1945 of *Lecture Notes in Computer Science*, pages 338–357. Springer-Verlag, 2000.

[Santos-Neto 08] Pedro Santos-Neto, Rodolfo F. Resende, and Clarindo Pádua. *An evaluation of a model-based testing method for information systems*. In *Proceedings of the 2008 ACM symposium on Applied computing*, SAC '08, pages 770–776, New York, NY, USA, 2008. ACM.

[Sarma 10] Monalisa Sarma, P. V. R. Murthy, Sylvia Jell, and Andreas Ulrich. *Model-based testing in industry: a case study with two MBT tools*. In *Proceedings of the 5th Workshop on Automation of Software Test*, AST '10, pages 87–90, New York, NY, USA, 2010. ACM.

[Shafique 13] Muhammad Shafique and Yvan Labiche. *A systematic review of state-based test tools. International Journal on Software Tools for Technology Transfer*, pages 1–18, 2013.

[Sinha 06] A. Sinha, C. E. Williams, and P. Santhanam. *A measurement framework for evaluating model-based test generation tools. IBM Systems Journal*, vol. 45, no. 3, 2006.

[Sommerville 10] Ian Sommerville. *Software Engineering*. Addison Wesley, 9 edition, March 2010.

[Staats 10] Matt Staats and Corina Păsăreanu. *Parallel symbolic execution for structural test generation*. In *Proceedings of the 19th international symposium on Software testing and analysis*, ISSTA '10, pages 183–194, New York, NY, USA,

2010. ACM.

[Sving 10] Robin Sving and Peter Öman. Pilot Project for Model Based Testing using Conformiq Qtronic. Master's thesis, University of Uppsala, 2010.

[Thorup 10] K. K. Thorup. *Triq*, 2010. https://github.com/krestenkrab/triq.

[Tretmans 96] J. Tretmans. *Test Generation with Inputs, Outputs, and Quiescence. Software – Concepts and Tools*, vol. 17, no. 3, pages 103–120, 1996.

[Tretmans 02] J. Tretmans and E. Brinksma. *Côte de Resyste – Automated Model Based Testing.* In M. Schweizer, editor, *Progress 2002 – 3^{rd} Workshop on Embedded Systems*, pages 246–255, Utrecht, The Netherlands, October 24 2002. STW Technology Foundation.

[Tretmans 03] Jan Tretmans and Ed Brinksma. *TorX: Automated Model-Based Testing.* In A. Hartman and K. Dussa-Ziegler, editors, *First European Conference on Model-Driven Software Engineering*, pages 31–43, December 2003.

[Utting 05] M. Utting. *Model-Based Testing.* In *Verified Software: Theories, Tools, Experiments*, Z rich, Switzerland, Oct. 2005. Position paper.

[Utting 07] Mark Utting and Bruno Legeard. *Practical Model-Based Testing: A Tools Approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.

[Utting 11] Mark Utting, Alexander Pretschner, and Bruno Legeard. *A taxonomy of model-based testing approaches. Software Testing, Verification and Reliability*, pages n/a–n/a, 2011.

[Vardi 86] M. Y. Vardi and P. Wolper. *An automata-theoretic approach to automatic program verification.* In *Proc. LICS '86, 1^{st} IEEE Int. Symp. on Logic in Computer Science*, pages 332–344, June 1986.

[Veanes 08] Margus Veanes, Colin Campbell, Wolfgang Grieskamp, Wolfram Schulte, Nikolai Tillmann, and Lev Nachmanson. *Model-Based Testing of Object-Oriented Reactive Systems with Spec Explorer.* In Robert Hierons, Jonathan Bowen, and Mark Harman, editors, *Formal Methods and Testing*, volume 4949 of *Lecture Notes in Computer Science*, pages 39–76. Springer Berlin / Heidelberg, 2008. 10.1007/978-3-540-78917-8_2.

[Vilkomir 01] S. A. Vilkomir and J. P. Bowen. *Formalization of software testing criteria using the Z notation.* In *Proceedings of the 25th International Computer Software and Applications Conference (COMPSAC 2001)*, pages 351–356. IEEE Computer Society Press, 8–12 October 2001.

[Vos 12] Tanja EJ Vos, Arthur I Baars, Felix F Lindlar, Andreas Windisch, Benjamin Wilmes, Hamilton Gross, Peter M Kruse, and Joachim Wegener. *Industrial Case Studies for*

Evaluating Search Based Structural Testing. International Journal of Software Engineering and Knowledge Engineering, vol. 22, no. 08, pages 1123–1149, 2012.

[Weiglhofer 09] Martin Weiglhofer, Bernhard K Aichernig, and Franz Wotawa. *Fault-Based Conformance Testing in Practice. Int. J. Software and Informatics*, vol. 3, no. 2-3, pages 375–411, 2009.

[Weißleder 10] Stephan Weißleder. *Test models and coverage criteria for automatic model-based test generation with UML state machines.* PhD thesis, Humboldt University of Berlin, 2010.

[Widera 04] M. Widera. *Flow Graphs for Testing Sequential* ERLANG *Programs.* In *Proc. 2004 ACM SIGPLAN workshop on Erlang*, Snowbird, Utah, USA, 2004.

[Wille 82] Rudolf Wille. *Restructuring lattice theory: an approach based on hierarchies of concepts.* In Ivan Rival, editor, *Ordered sets*, pages 445–470, Dordrecht–Boston, 1982. Reidel.