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

Johan Blom
Blossom Grove AB
Email: johan.blom@blossomgrove.se

Bengt Jonsson
Uppsala University
Dept. Information Technology
Email: bengt.jonsson@it.uu.se

Sven-Olof Nyström
Uppsala University
Dept. Information Technology
Email: sven-olof.nystrom@it.uu.se

***Abstract*—We report on a case study on model based testing for a commercially available telecom software system. A main purpose is to investigate how different strategies for test suite generation affect quality attributes of the generated test suites, in a realistic industrial environment. We develop a functional model in the form of an extended finite state machine, from which we generate test suites using several different (model) coverage criteria, alongside with randomly and manually generated test suites. We compare test suites with respect to fault-detection capability, incurred (source) code coverage, and test generation and execution time. The system under test is a commercially released version, not seeded with any faults, implying that exposed faults are "real" faults that passed previous testing. We did not find clear difference between coverage-based and random test suites. Test suite generation and execution is performed using the tool ERLY MARSH, developed by the first author.**

I. INTRODUCTION

Model Based Testing aims to make testing more systematic and more automated. It has been applied to the testing of control-intensive software applications, such as communication and control software in various embedded systems domains [1], [2]. The basic idea in model based testing is to construct a formal executable model, which captures the intended behavior of the system under test. From this model, test cases can be systematically generated as execution traces. Since the model is formal, test generation can be automated. With adequate tool support one can automatically generate very large test suites. There are several academic and commercial tools that support modeling and model based testing, including ConformiQ Qtronic [3], JTorX [4], and Spec Explorer [5].

Given available tool support for automated generation of test suites, an obvious question is how test suites should be generated to optimize important quality attributes, such as capability to detect faults, cost of generation and execution, etc. In a given situation, the best trade-off between these partly conflicting attributes depends on the desired level of quality, the cost (time) of executing each test case, the available test environment, etc. It is therefore important to understand how different strategies for test suite generation affect quality attributes of test suites. Test suites can be generated, e.g., using different (model) coverage criteria, using random input generation, from manually supplied test purposes, or manually. They can be evaluated based on their size, capability to detect faults (implementation defects), provoke failures, incur (source) code coverage, time needed for execution, etc. For white-box structural testing, these issues have been well studied, e.g., concerning the relationship between code coverage and fault detection capability (e.g., [6], [7]). For model based testing, which is a form of black-box functional testing, the corresponding relationships have been studied to a significantly lesser degree. There are some case studies on industrial safety-critical systems, which use mutated faults to evaluate detection capability (e.g., [8], [10]), with not so clear results. We are not aware of such studies on industrial software that use "real" bugs, as opposed to seeded ones. One reason is that such studies need a significant effort for building functional models, and for evaluating test results.

In this paper, we investigate the relationship between test suite generation strategies and test suite quality attributes, for a real industrial benchmark. The investigation is performed as a case study on model based testing of a commercially available telecom software system, Mobile Arts Advanced Location Center (A-MLC). As System Under Test (SUT) we used a version of A-MLC which had been manually tested, but still contained a significant number of faults ("bugs"). We did not seed the SUT with any faults. Thus, all faults found are "real" faults that occur in actual industrial software development. We developed a model of the intended functionality of A-MLC in a formalism for extended finite-state machines [11]. From this model, we generated test suites of different sizes and with different strategies. Some test suites were generated using different (model) coverage criteria, some were generated randomly, and some were manually generated. We applied each test suite to the SUT and measured resulting quality attributes, including size, number of exposed failures, number of exposed faults, (source) code coverage, and time needed for test suite generation and execution. We then analyzed the results to better understand the relationship between test suite generation strategies and test suite quality attributes. Test suite generation and execution was performed using the tool ERLY MARSH, developed by the first author.

As mentioned, the case study is performed on A-MLC. A-MLC is to a large extent implemented in the ERLANG programming language. The first author has contributed to its development, and we had access to a sequence of snapshots of its implementation. As a basis for testing, we developed

a model of its functionality in the modeling language ERLANG/EFSM, which is a formalism for extended finite-state machines (EFSMs) [11], [12], where guards and actions are described in ERLANG syntax. To support model based testing based on ERLANG/EFSM models, we have developed a tool chain, ERLY MARSH, which supports automated test generation, test execution, and test report generation. The tool ERLY MARSH supports specification of a range of test suite coverage criteria, described using a formalism based on observers [13]. It employs symbolic execution techniques to generate test suites, with coverage criteria specified by observers. From the ERLANG/EFSM model of A-MLC, we used ERLY MARSH to generate differently sized test suites satisfying various (model) coverage criteria. For comparison, we also generated random test suites of similar size. We also had access to a manually generated test suite, developed prior to our modeling effort.

We evaluated the test suites according to quality attributes, including size, number of exposed failures, number of exposed faults, (source) code coverage, and time for test suite generation and execution. In the evaluation, particular emphasis was given to the number of faults detected. As system under test (SUT), we used a snapshot from the development of A-MLC, which had been manually tested, but could still be expected to contain faults. We did not seed the SUT with any artificially generated faults. Our ambition to use "real" faults forced us to trace the cause of each exposed failure in the source code. This sometimes required significant effort. In order to understand precisely which faults were exposed by a particular test suite, we developed a mechanism of applying bug fixes, for automatically detecting which faults are revealed by a particular test suite. This means that our evaluation measures the test suites' capability to detect faults that occur in industrial settings.

With the ERLY MARSH tool for test suite generation and the bug fix system for counting faults in place, we could then proceed with the experiment. We start from the hypothesis that test suites can have a measurable quality. We executed each test suites on the implementation, in order to measure its quality. which can be controlled by techniques for generating test suites. We were particularly interested in questions such as

- the relationship between test suite size and fault detection capability for coverage-based, random, and manual test suites,
- comparing the fault-detection capability of coverage-based, random, and manual test suites, for comparable test suite sizes,
- the relationship between (model) coverage, test suite size, and implementation code coverage, and
- to understand which kinds of faults are revealed by model based testing, in comparison with faults detected by standard manual testing.

Following [14], we organized the evaluation by letting parameters that control the generation and execution of test suites be *independent variables* and the observable effects, such as number of detected errors, be *dependent variables*.

We note that using (model) coverage criteria to generate test suites yields a good trade-off between fault-detection capability and test suite size, for moderate-to-large size test suites. A Particularly favourable relationship is exhibited by the All-Edges criterion. We also note that a more detailed coverage criterion detects more faults than a less detailed one. For very large test suites, the coverage is so complete that the difference between different test strategies (for example, coverage based vs. random) is not so prominent. Another finding was that, at least in our case study, model-based testing discovered faults also in previously well-tested parts of the application, partly because previous testing had involved a relaxed validation of requirements. These findings complement findings by other studies, as discussed in Section V.

Since our evaluation is performed on only a single SUT, it is of course not clear how well our results generalize to other software systems. Since a case study of this form is a very time-consuming endeavour, where significant effort is required for building a model and for tracing faults, it was not possible for us to perform similar experiments on other software modules. We hope that the literature on related case studies will expand to allow better understanding of strategies for model-based test suite generation.

Notes on terminology We define a *failure* to be an observed deviation between the behavior of the system under test (SUT) and the A-MLC model. We define a *fault* to be a cause of a failure. A modification to the SUT to remove a fault is referred to as a *bug fix*.

Outline The paper is organized as follows. Section II presents the setup of the case study surveying the SUT, the ERLANG/EFSM specification of the SUT, and the test execution environment, Section III contains a description of how faults are quantified using our framework of bug fixes. In Section IV, we describe the independent variables, which are test suite generation strategy (coverage-based, random, or manual) and the dependent variables, which are number of exposed failures, number of exposed bug fixes, source code coverage, test suite size, and execution time. Main results of the experiment are summarized in Section V, which also contains a summary of the main findings. We survey related work, in particular on similar industrial case studies, in Section VI, we report on related case studies in the literature. Conclusions are summarized in Section VII.

II. MOBILE ARTS ADVANCED MOBILE LOCATION CENTER

Mobile Arts A-MLC (Advanced Mobile Location Center) is a middle-ware product to allow Mobile Network Operators to provide presence data (details about the location, current status and capabilities) about mobile devices. For example, an emergency center (i.e., a presence dependent application) may want to know where a calling user is located, in order to send the closest available ambulance to the caller. Applications using A-MLC communicate via MLP [15], an XML based protocol, over HTTP. A-MLC in turn uses the SS7 protocol

stack to communicate with GSM/UMTS/LTE core network. See Figure 1 for an illustration. Since AMLC is important for many applications and the requirements on availability and fault tolerance are high, careful testing is necessary. A-MLC is commercially available and has been deployed with several mobile network operators in Europe and Asia.

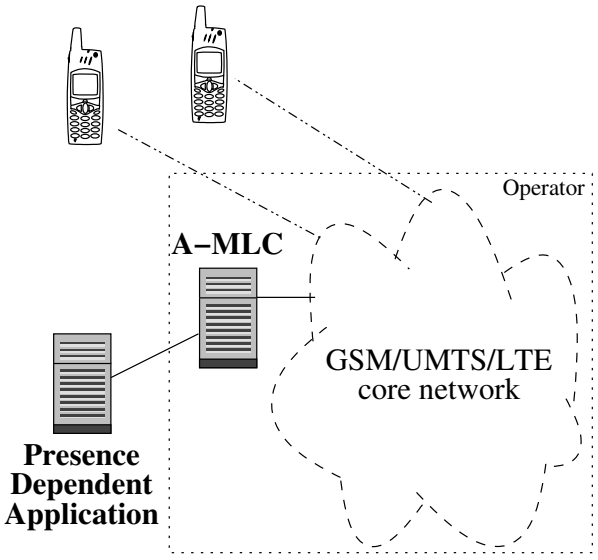

Fig. 1. An outline of how Mobile Arts A-MLC can communicate with a presence dependent application and mobile devices using a number of GSM/UMTS/LTE core network nodes.

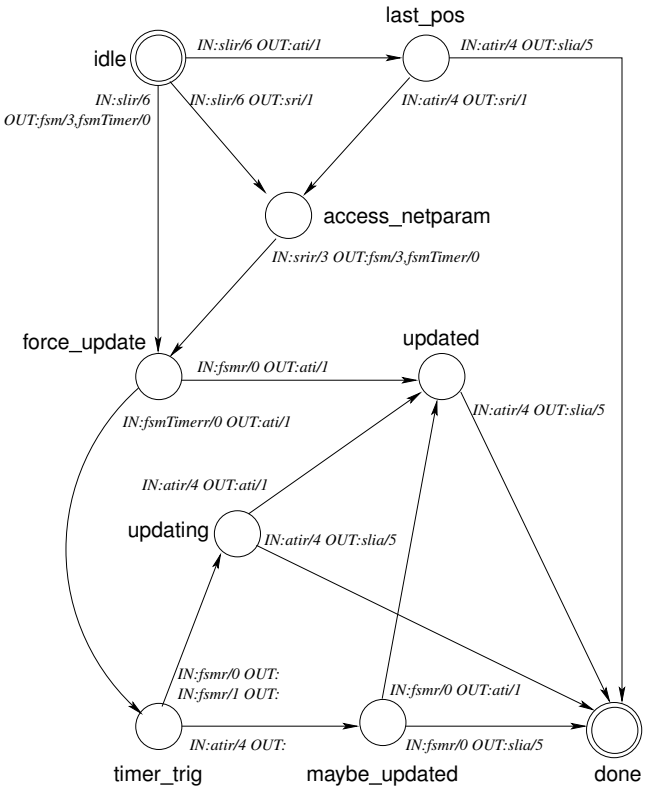

Fig. 2. A graphical representation of a part of the state machine created for A-MLC. Note that due to space restrictions some edges are omitted.

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 standard 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. Still, the functional specification evolved simultaneously with the implementation, sometimes creating problems to keep the system consistent. Development was also characterized by lack of resources for testing. It was therefore decided to formalize the functional specification and complement a manually created (hand-crafted) test suite with a test suite based on the formalization.

Before the evaluation A-MLC had been tested in cooperation with a customer and deployed. After deployment followed development of new functionality before a snapshot was made for this evaluation.

A. *The A-MLC* ERLANG/EFSM *specification*

As a specification language of the SUT we used ERLANG/EFSM [12], due to its ability to express communication behavior at a high level of abstraction. ERLANG/EFSM is based on the functional language ERLANG and retains features such as pattern matching, dynamic typing, and single assignment variables. For the specification of *state machines*, ERLANG/EFSM additionally includes constructs for control locations, state variables, and structured messages.

The (iterative) process of creating the formal specification took a significant amount of effort. Interpreting the functional specification and creating an initial model was a tedious work, caused by the detailed functional specification containing unclear, wrong, or missing information. Additionally, after executing a test suite, a revealed failure often could be attributed to an error in the created formal specification.

The A-MLC ERLANG/EFSM specification covers all of the functionality provided by the detailed functional specification. Interfaces between A-MLC and its environment (the presence dependent application and the core network) are implemented with standard protocols, modeled at the application level. Only relevant messages in these standard protocols were modeled. The reaction to non-well-formed messages, was also not modeled, since it is assumed that they are rejected directly by the interface. Likewise, the operation and maintenance interface (for example: counters, alarms, and GUI) is omitted.

The A-MLC ERLANG/EFSM specification also includes a number of configuration parameters to represent subscriber- and session-specific data, such as preferred billing method. These parameters are retrieved from a database at session initiation.

The resulting A-MLC ERLANG/EFSM specification captures all possible traffic sequences through A-MLC towards a

presence dependent application, and all relevant operations towards the GSM/UMTS/LTE core network. In total, the model contains 12 locations, 15 configuration parameters, 11 input event types and 8 output event types where each event type have between 0 and 6 event parameters. In total, the model contains 85, mainly boolean, symbolic input parameters.

The model represent an abstraction of the actual behavior of the A-MLC. For example, status codes are abstracted to a few equivalence classes and collected age information on a location is abstracted into "true" if acceptable and "false" otherwise.

The model represents the behavior of a single session of the protocol. It does not model several concurrent sessions, since ERLANG provides native support for concurrency and fault isolation between threads.

Figure 2 outlines a part of the state machine created for A-MLC. Execution starts in idle when an incoming *slir/6* occurs. This message includes parameters to control quality of present data returned, request positioning method and provide cached address data. In particular it is possible to request the last cached, or current, presence data. Assuming we want the current position data we must first force an update by sending a *fsm/3* and move to the force_update location. Now assuming the response *fsmr/0* was received in a timely manner we can request the presence data with a *ati/1* message and move to updated. Finally, when *atir/4* is received we are done and can respond with a *slia/5* message back to the presence dependent application.

B. Test Suite Generation

ERLY MARSH is a tool for model-based test suite generation and test suite execution. that we use to generate test suites from from ERLANG/EFSM models. In principle, we can generate test cases as executions of the ERLANG/EFSM model. Since the ERLANG/EFSM model is an abstraction of the actual behavior of a SUT, it follows that the generated test cases will be at the same level of abstraction. We use the term *abstract test case* for such test cases. Actual (concrete) test cases can be generated from abstract test cases by appropriate concretization/abstraction functions that are applied during test execution.

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 such executions. However, this approach will run into the well-known state-space explosion problem for large test suites. We therefore use a symbolic approach, in which symbolic execution is applied to the ERLANG/EFSM model to generate so-called *symbolic test cases*. A symbolic test case essentially consists of a path in the ERLANG/EFSM model and a symbolic path constraint on the parameters of occurring input messages. A symbolic test case compactly represents the set of abstract test cases that can be obtained by assigning values to input parameters that satisfy the path constraint. For compact representation of symbolic test cases we use NDDs (*Numerical Decision Diagrams* [16]), since many parameters in the A-MLC ERLANG/EFSM specification range over small finite domains.

We provide a mechanism for specifying coverage criteria using observers [13]. Given an observer and a specification, a test suite can be generated automatically. Essentially, observers can be seen as an instrumentation of a ERLANG/EFSM model, given in a defined syntax, which monitor the execution of test cases and report desired properties, such as contributing to some coverage criterion. Observers can in principle specify any coverage criterion, and are therefore a versatile tool for generating a range of different test suites.

Input events in the abstract test cases in an abstract test suite are concretized by a SUT specific call-back module, which replaces abstract parameter values by concrete ones. Similarly, abstraction of concrete events to abstract output events are handled by the same call-back module.

C. Test Execution and Verification

For efficient execution of a generated test suite, we must set up a test harness. When testing the A-MLC, this is complicated by the fact that the test harness must handle concurrent test cases. In order to identify which test case each incoming message belongs to, each test case is assigned a test case identifier that is embedded within the messages exchanged between the SUT and its environment.

Messages to/from the core network are simulated by utilizing ERLANG remote procedure calls between the SUT and the test harness. Messages to/from the presence dependent application utilize full implementations of that protocol.

After the test case has finished executing, 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 a HTML based user interface.

Use of configuration data, requires the SUT to handle many concurrent configuration data sets with concurrent execution of test cases. Fortunately A-MLC supports concurrent execution.

III. MEASURING FAILURES AND FAULTS

One goal of our case study is to evaluate test suites by the number of detected errors. Moreover, we intend the occurring errors be errors that remain after some amount of testing in a realistic industrial project. Therefore, our SUT was not seeded with any faults. All occurring faults are "real" faults, not (yet) detected by any other kind of testing. In our view, this allows a more realistic evaluation of test suites. It also implies that the errors were not known to us before the case study. Since our testing is black-box, a test suite can only reveal observable effects of errors, not the mistake in the source code that caused the error. We therefore had to trace down the causes of all observed failures manually. In order to quantify the number of errors revealed by each test suite, we developed a technique for automatically determining the number of programming errors that are exposed by a test suite or test cases, in the form of bug fixes.

When executing a concrete test case we assume a black-box view of the SUT and therefore initially only observe failures

(when bug-fixes are introduced they can also be observed). Recall that a *failure* is defined to be an observed deviation between the behavior of the SUT and the A-MLC model and 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, the existence of a failure indicates a need to fix one or more faults.

In our experiments we wish to study programs containing multiple faults. This causes additional difficulties as the faults may be dependent. Thus we cannot decide which faults a particular test case reveals as the presence of one fault may prevent another from being detected. Our approach to this problem is to construct a bug fix for each fault that was revealed by some observed failure. We also developed a system for applying any combination of these bug fixes to the SUT, and rerunning a test suite to see whether failures disappeared. Then we measure the number of faults detected by a test suite as the number of bug fixes needed to remove the failures exposed by the test suite. Thus we use bug fixes as a way to characterize and quantify faults.

Let us provide some more details on the granularity of bug fixes. 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. A fault must originate in some executable statement. Thus, we define a bug fix as a change of an executable statement, to which a fault can be attributed, into a correctly behaving ERLANG expression.

We used source code coverage as another measure of test suite effectiveness. To measure source code coverage we used the COVER tool [17], which measures source code coverage of ERLANG programs. COVER counts how many times each executable line of source code is executed when a program is run. An line is considered to be executable if it it contains an ERLANG expression such as a matching, guard or a function call, but lines only containing a comment or a pattern are not considered to be executable.

IV. PREREQUISITES FOR CASE STUDY

Our case study is based on the hypothesis that test suites may be of a measurable quality, and that the quality can be controlled by techniques for generating test suites. Inspired by Rothermel et al. [14], we therefore organized the evaluation by defining *independent variables*, i.e., parameters that control the generation and execution of test suites, and *dependent variables* that measure the quality of a test suite, called observable effects, such as the measured quality of a test suite, of the different values of the independent variables.

The independent variables are (A) Test case selection techniques and (B) Test execution strategies. The dependent variables are (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). In the following subsection we discuss the independent variables in more detail, the dependent variables are further discussed in Section V.

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).

A. Test case selection techniques

In the evaluation, we generated test suites satisfying different coverage criteria. For comparison, we also included a number of randomly generated test suites of similar sizes, and a hand-crafted manual test suite.

Coverage-based test suites were generated using the following criteria:

- *All paths* (All-Paths) cover all possible paths by making an exhaustive unfolding of the branching structure of the model.
- *All locations* (All-Locs) cover each location in the source code at least once.
- *All edges* (All-Edges) cover each possible edge at least once; here and "edge" correspond to an exit point of some transition in the ERLANG/EFSM model.
- *All Definition-Use pairs* (Def-Use) cover all possible pairs of definition and usage of a pair of occurrences of a state variable in the ERLANG/EFSM model.

Each test suite was generated by first generating a suite of symbolic test cases, from which an abstract test suite with the chosen coverage can be obtained by instantiating the symbolic parameters (i.e., input parameters and configuration data) with values that satisfy the corresponding path constraint. In the evaluation, the parameter values were chosen at random from the set of possible satisfying assignments, using a uniform distribution.

Randomly generated test suites are intended to emulate a test execution strategy in which the next input message to A-MLC is chosen at random with a uniform distribution. Since, at each point during test execution, a vast majority of input messages are unspecified, it would lead to poor results to perform random testing by just supplying a sequence of random input messages. Instead, we let the next input message be chosen at random from the set of *legal* inputs at that stage of test execution. An input is legal if its effect is specified in the A-MLC specification. Thus, the random test suites can be said to emulate a "model-based random test selection", in which test selection uses the A-MLC model to obtain the set of next legal inputs and choose a random next input following uniform distribution. We first generate the symbolic test suites. To obtain the effect of selecting among the next input messages according to a uniform distribution we assign a weight to each symbolic test case, proportional to the probability that a randomly generated abstract test case is an instance of this symbolic test case. Thus abstract test cases are selected 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 among those covered by the selected symbolic test case. In order to make this selection correspond to a selection in actual random test generation, we assigned a *weight* to each symbolic test case, which

is proportional to the probability that a randomly generated abstract test case (according to the distribution obtained by selecting each input at random from the set of legal inputs) is an instance of this symbolic test case.

For comparison, we also created random test suites, where the selection of test cases used a uniform distribution over all feasible abstract test cases. This distribution typically gives a strong bias towards long test cases with many possible values of input parameters.

Manually generated test suites At the time the evaluated snapshot of A-MLC was made, not all of the traffic scenarios found in the functional description had been deployed at a customer. Therefore, the test suite used at Mobile Arts for testing A-MLC was limited to 20 test cases, and covered 6 out of 11 traffic scenarios defined in the functional specification, representing the most frequently occurring use cases. We will refer to this limited test suite as Man_{lim}. Manual test cases for the additional traffic scenarios, were constructed by the first author following the same patterns as in the Man_{lim} test suite. The full manual test suite is referred to as Man. Only a few of these test cases represented error cases.

For the evaluation, these test cases were translated into a format suitable for ERLY MARSH so that automatic test case execution, including validation, could be applied. As with the Randomly generated test suites above, we then first generated the symbolic test suite with all symbolic test cases (All-Paths), and then selected matching abstract test cases.

B. Test execution strategies

Since the execution of one test case entails waiting for (for example) I/O and time-outs, concurrent execution of the test cases gives a significant speed up, even on single-core platforms. Each test suite was executed both sequentially (only a single ERLANG process in the execution of the test cases could run at any time), and concurrently. Concurrent test execution was made significantly easier by ERLANGs support for concurrency and fault isolation.

V. SUMMARY OF EXPERIMENTAL RESULTS

In this section, we summarize the results of our case study. The purpose of the experiment is to investigate how different test generation and test execution strategies, represented by independent variables, influence different properties of test suites (i.e., failures, faults, source code coverage, abstract test suite size, and execution time), represented by dependent variables. Table I summarizes the results for coverage-based test suite generation, and Table II summarizes the results for randomly generated test suites. In these tables, we use the following terminology.

- Failures is the number of different failures exposed,
- Bug fixes is the number of bug fixes required to correct detected faults; we use this as the number of exposed faults,
- Code Coverage is the fraction of lines of code covered for selected ERLANG modules on the SUT, after excluding code out of scope of the functional model,

TABLE I
SUMMARY OF TEST RESULTS FOR COVERAGE BASED TEST SUITES GENERATED FOR A-MLC.

Test suite name:	All-Locs	All-Edges	Def-Use	All-Paths
Symbolic TC	5	422	6292	82423
Failures	2	12	12	15
Bug fixes	13	78	83	96
Code cov	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	00:00:01	00:00:01	00:00:10	00:13:26
Time exe seq	00:00:04	00:06:50	03:54:23	62:00:40
Time exe con	00:00:01	00:00:12	00:03:18	00:55:33

- Test Cases is the number of test cases executed,
- Time gen symb is the time measured for generating the symbolic test suite,
- Time gen abs is the time measured for generating the abstract test suite from the obtained 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.

Table I shows the results for coverage based test generation strategies, for the coverage criteria All-Locs (*All locations*), All-Edges (*All edges*), Def-Use (*All Definition-Use pairs*), and All-Paths (*All paths*). Since, as described in Section II-B, we only select one abstract test case from each symbolic test case, and one concrete test case from each abstract, the number of *executed* test cases is identical to the number of symbolic test cases in the test suite. On the other hand, it is interesting to note, that for our model of A-MLC, there are around $4 * 10^{11}$ *possible* abstract cases that can be generated from the $82,423$ symbolic test cases.

Table II shows the results for a selection of randomly generated test suites. The notation in the top row is that Rnd_N^w is a random test suite emulating a distribution of type w (where ew denotes a distribution obtained by selecting each next input uniformly among legal inputs and tcw denotes a uniform distribution over all $4 * 10^{11}$ abstract test cases), with N test cases. Here Symbolic TC denotes the number of different symbolic test cases that are "covered" by the random test suite, essentially the number of covered paths referred to by the All-Paths coverage criterion. The reported results are the mean values of three independently created test suites. Random test suites were created for several different sizes, for brevity only a subset are reported in Table II.

The results of running the manually generated tests suites are shown in Table III.

TABLE III
SUMMARY OF TEST RESULTS ON THE A-MLC FOR MANUAL TEST SUITE.

Test suite name:	Man	Man_{lim}	All-Paths^{red}
Test cases	50	20	581
Failures	2	2	7
Bug fixes	31	15	41
Code Coverage	80.98%	66.70%	66.33%

TABLE II
SUMMARY OF TEST RESULTS FOR CREATING AND EXECUTING A SELECTION OF THE RANDOM TEST SUITES ON THE A-MLC.

Test suite name:	Rnd^{tcw}_{1k}	Rnd^{tcw}_{12k}	Rnd^{tcw}_{48k}	Rnd^{tcw}_{100k}	Rnd^{ew}_{1k}	Rnd^{ew}_{12k}	Rnd^{ew}_{48k}	Rnd^{ew}_{100k}
Test cases	1000	12,000	48,000	100,000	1000	12,000	48,000	100,000
Symbolic TC	968	8,464	20,276	28,583	313	1195	2361	3283
Failures	12	13	15	15	13	15	15	15
Bug fixes	48	53	57	60	83	98	98	98
Code Coverage	83.08%	86.84%	89.13%	90.76%	96.20%	97.58%	97.55%	97.79%
Time exe con	00:00:46	00:09:02	00:31:59	01:02:00	00:00:14	00:02:56	00:11:01	00:23:55

We were initially surprised by the large number of bug fixes exposed by Man_{lim}, considering that it had previously been applied to an earlier version of the SUT. After investigations, we concluded that the main reason was that previous testing at Mobile Arts had been concentrated to limited parts of the SUT, and that the values of several parameters in output events had not been validated during testing.

In order to better understand the coverage obtained by the manual test suite, we created a *reduced model* from the original specification, by excluding functionality which had previously been incompletely tested, and relaxing the validation of parameters in output events. From this reduced model, we derived a test suite All-Paths^{red} with full path coverage, using ERLY MARSH. The results of applying All-Paths^{red} are also included in Table III.

A. Investigation into Exposed Failures

During test suite execution, 15 different failures were exposed. Of the 15 failures, 13 failures consisted in a deviation in the output generated by the SUT and the output expected by the model. That is, the SUT either did not generate an output although one was expected, generated the wrong output type, or generated an output although none was expected. Two of these failures were deviations in parameter values. The effect of these deviations mostly concerned the quality of reported data: for example, the actual quality of returned presence data could be incorrectly reported (the age reported was more recent than the actual age), additional unnecessary requests for updated presence data would be performed. Thus, we judged these deviations to have low or middle severity. The two remaining failures were of high severity. One failure was essentially a memory leak, which would cause the SUT to crash after some time, and one failure was an uncaught exception, which would cause the concerned session to crash without notifying the environment.

Different test suites found different subsets of the failures and only the largest, most complete test suites, found all. It can be noted that the (minimal) All-Locs test suite did not find any of the highly severe failures. The All-Edges test suite found almost as many failures as the Def-Use test suite, despite that test suite included 93% less executed test cases. However, it did not find the severe memory leak, since this required the size of the test suite to be at least 1000 test cases.

B. Investigation into Exposed Faults

The number of different exposed faults, measured as bug fixes, is reported in Tables I, II, and III. In total we found 102 different bug fixes exposed by some test suite. No test suite found all of these bug fixes.

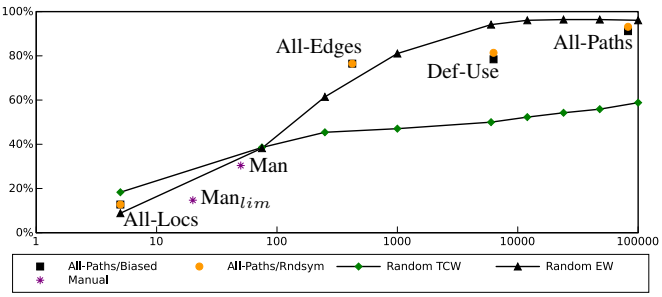

Fig. 3. Relative number of different bug fixes exposed when running the coverage based test suites with the original base model and biased and random abstract test case selection, random test suites generated using *tcw* distribution (Rnd^{tcw}_{100k}), and random test suites generated using *ew* distribution (Rnd^{ew}_{100k}). The x-axis shows the number of test cases executed, and the y-axis shows the percentage of exposed bug fixes (of all 102) that were exposed.

In order to compare test suites by their relation between cost and benefit, we related the percentage of all 102 bug fixes exposed by a test suite with the size of that test suite, measured as the number of test cases. Figure 3 summarizes the results for the coverage based test suites, a number of random test suites, and the manually generated test suite. As can be seen in Figure 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 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 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. Also, the Man test suite exposed relatively few bug fixes.

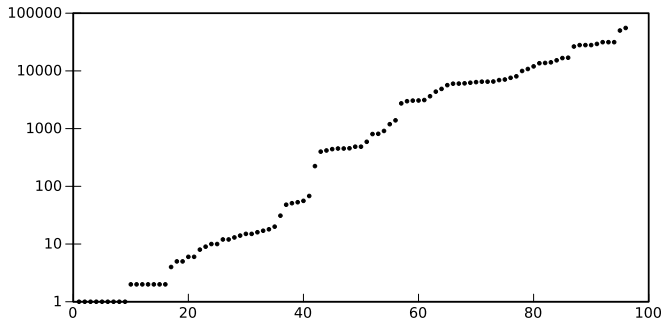

Fig. 4. Number of test cases exposing each bug fix when running the All-Paths test suite with random abstract test case selection. The x-axis shows bug fixes, each enumerated with a unique integer, and the y-axis shows the number of test cases exposing a bug fix.

In order to understand whether some faults were harder to detect than others, Figure 4 shows how many times each bug fix was exposed when executing the All-Paths 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. On the other hand, a majority of the bug fixes are exposed by more than a thousand test cases. The most easily detected bug fix was exposed by $55,446$ test cases.

We also made a further investigation into *when*, during the execution of a test suite, new bug fixes were exposed. Due to the quite different results, with respect to exposing bug fixes, of the two types of random test suites in Table II we expected test case length to influence the results. To investigate, we executed three of the larger test cases (All-Paths,Rnd$^{ew}_{100k}$,Rnd$^{tcw}_{100k}$) in order of increasing length. After each test case finished executing, we noted the total number of bug fixes exposed by the test suite and compared these results with unsorted test suites. From these executions it was clear that a sorted All-Paths test suite was clearly most efficient on exposing bug fixes out of the compared test suites. Reason for this is likely that there exists faults in many different parts of the SUT. The All-Paths 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.

C. Code coverage

Code coverage of the SUT was measured using the COVER tool on a set of ERLANG modules, 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 modules, including e.g., test functions and debugging utilities. After investigation, we excluded code that could not be exercised by our specification, such as unreachable code, diagnostic code, etc. We then renormalized the code coverage in relation to the test suite with the reported highest coverage (All-Paths). The results are shown in Figure 5, presented in a way analogous to Figure 3.

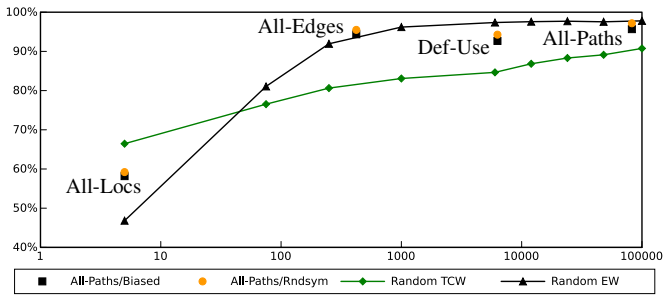

Fig. 5. Relative source code coverage when running the random test suites and the coverage based test suites with the original base model. The x-axis shows the measured relative coverage and the y-axis shows the number of test cases executed.

It can be seen that the correlation between test case size and code coverage is similar to that for number of exposed faults. In particular, All-Edges stands out as yielding high code coverage.

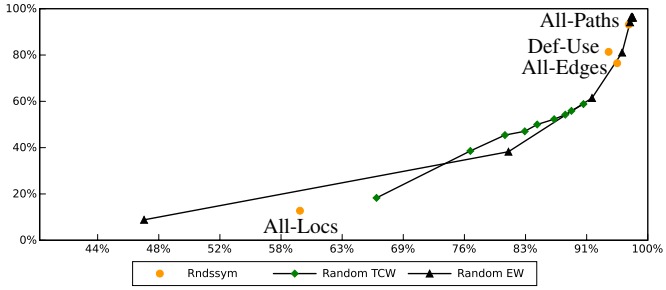

Fig. 6. Relation between source code coverage and number of exposed bug fixes. with the original base model. The x-axis shows the measured relative coverage and the y-axis shows the percentage of exposed bug fixes.

Figure 6 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 and All-Edges perform comparatively well with rather limited sizes of the test suites.

D. Test Generation and Execution Times

We also briefly comment on the times needed for generation and execution of test suites as reported in Tables I and II. One observation is that the generation of the test suite Def-Use takes significantly longer than the generation of, e.g., the larger test suite All-Paths. The explanation is that our generic mechanism for handling coverage, using observers, incurs a significant overhead, in particular for this coverage criterion. Another observation is that concurrent test execution saves up to two orders of magnitude in total test execution time. Such differences are especially prominent for test cases that involve timeouts before the test case can continue, which is quite

common in test suites for A-MLC. 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.

E. Summary of Main Findings

Let us here summarize main observations and findings from the results of the case study.

- The test suite All-Edges had a comparatively good relationship between number of exposed faults and test suite size; a stronger coverage criterion increases the fault-detection capability marginally.
- We found no clear difference in power between coverage-based test suites and randomly generated test suites of the same size, as measured by the number of detected faults and source code coverage.
- As the size of test suites increase, the differences between different test suites decreases.
- Traditional manual testing often does not check that test outputs correspond completely to requirements. Formalizing requirements by a model, and using it for automated testing, can reveal deficiencies of earlier "sloppy" testing. We note that this observation need not generalize to other contexts.

During the evaluation, we found that tool support is essential for effective test suite generation, test execution, and model validation. The observation that at least one severe failure was dependent on the size of the test suite indicates that testing functional requirements by model-based testing should be complemented with other types of testing for finding resource and other non-functional errors, such as memory leaks.

F. Relationship with Other Studies

Let us also compare our findings regarding the power of coverage-based test suites with findings from other related studies. Such studies include [8], [10], which differ in that faults are injected by mutation. Heimdahl et al. [8] generate coverage-based test suites using a different technique (model checking) than in our tool. This results in an abundance of very short test cases, with a modest ability to detect faults. They explain the poor fault detection ability by observing that the structure of the system does not well match the coverage criterion, and that criteria based on decision coverage may be deficient if they to not force the test suite to exercise relevant data flows. The findings by Heimdahl et al contrast with the findings from our case study, in which the coverage-based test suites perform comparatively well. Contrasting with [8], we note that our test generation technique, using observers, does not have an excessive bias towards short test cases, and that the data flow coverage criterion of Def-Use gives excellent fault detection power. Poor performance of only coverage-based test suites was again observed in [10]. Reading this, one possible explanation is that their counterexample-based technique for test suite generation creates some bias. It could be the case that our observer-based technique avoids such bias, but deeper investigations are needed before this can be confirmed.

VI. Related Work

The literature on model based testing in general is vast, covering, e.g., supporting techniques for modeling and test generation (e.g., [18]), tools (e.g., [20]) and integration into industrial practice (e.g., [22]).

Several works report on industrial case studies with the aim of comparing it to existing traditional testing approaches. Among these, the study [23] reports that Spec Explorer is used extensively internally at Microsoft; among preliminary results it reports that the application of model based testing techniques to 125 protocols with an investment of 50 person years has resulted in a 42% productivity gain when compared to traditional (manual) testing. Another industrial case study [24], at Siemens, compares Spec Explorer [5] to ConformiQ Qtronic [3]; it concludes that these tools deserve to be considered in industrial projects and identifies some shortcomings, such as integration with different types of testing (e.g., performance testing and unit testing), possibilities to optimize test suites (e.g., by prioritizing test cases and designing better coverage criteria), and support for round-trip engineering (including tracing of faults). Pretschner et al. [25] compare model-based testing with manual testing. They report that test suites generated from a model found significantly more requirements faults than manual testing, while the number of detected programming faults was approximately equal.

In the literature, we did not find too many works that evaluate different strategies for test suite generation in model based testing. Many of these evaluate test suites using seeded faults. Five different case studies are reported in [26], with test suites ranging from 10 to several hundred test cases. These case studies use a UML specification 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 in the industrial case study presented by Heimdahl et al. [8]. They find that coverage-based test suites can be poor in detecting faults, and that randomly generated test suites may perform better. This study and subsequent work, partly by the same authors [10], contrast to our findings, as discussed in Section V-F.

In [27], compares a technique called fault-based testing with random and manual testing. Fault-based testing is designed to detect certain specified faults in an implementation, and the case study confirms that the technique is indeed more effective than other techniques.

For white-box testing without a model, there are several comparisons between coverage-based and random testing. As an example, [28] compares coverage based testing with random testing on declarative programs. The results indicate that random testing is quite ineffective for some benchmarks, while coverage based techniques catch faults more consistently.

VII. Conclusions

We have reported from a case study on model-based testing. The main goals of the case study was to investigate how different quality attributes of test suites, generated from a

formal model, depend on the strategy used for their generation. Our intention was to perform this investigation under realistic industrial conditions. We therefore chose as our object of study an industrially deployed software component of significant size, which had been subjected to traditional testing. Moreover, we wanted detected faults to be faults that typically remain after some amount of traditional testing. Thus, the case study was not seeded with any faults: all faults in the test software were "real" faults, not (yet) detected by traditional testing.

Among our findings was that using (model) coverage criteria to generate test suites yields a good trade-off between fault-detection capability and test suite size, for moderate-to-large size test suites, which outperforms that of randomly generated test suites. Another finding was that, in our case study, model-based testing discovered faults also in previously well-tested parts of the application, partly because previous testing had involved a relaxed validation of requirements. We could validate findings by other works concerning the prerequisites and advantages of model based testing, such as the importance of automated tool support, including support for presentation of test results, support for manually defining additional test cases, etc. The literature also contains studies with findings that are not consistent with those of ours, although there are differences that may plausible explain the differences in findings.

Performing a case study of this form requires a significant effort for modeling building and tracing of faults. Thus, it has not been possible to repeat this experiment for other software modules, implying that our results do not necessarily generalize to other software modules. We hope that the literature will provide additional reports on case studies that will lead to a more solid understanding of strategies for test generation in model based testing.

REFERENCES

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

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

[3] A. Huima, "Implementing Conformiq Qtronic," in *Testing of Software and Communicating Systems*, ser. Lecture Notes in Computer Science, A. Petrenko, M. Veanes, J. Tretmans, and W. Grieskamp, Eds., vol. 4581. Springer Berlin / Heidelberg, 2007, pp. 1–12, 10.1007/978-3-540-73066-8_1. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-73066-8_1

[4] A. 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*, ser. TACAS'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 266–270. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-12002-2_21

[5] M. Veanes, C. Campbell, W. Grieskamp, W. Schulte, N. Tillmann, and L. Nachmanson, "Model-based testing of object-oriented reactive systems with Spec Explorer," in *Formal Methods and Testing*, ser. Lecture Notes in Computer Science, R. Hierons, J. Bowen, and M. Harman, Eds. Springer Berlin / Heidelberg, 2008, vol. 4949, pp. 39–76, 10.1007/978-3-540-78917-8_2. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-78917-8_2

[6] H. Zhu, P. Hall, and J. May, "Software unit test coverage and adequacy," *ACM Computing Surveys*, vol. 29, no. 4, pp. 366–427, Dec. 1997.

[7] J. Andrews, L. Briand, Y. Labiche, and A. Namin, "Using mutation analysis for assessing and comparing testing coverage criteria," *Software Engineering, IEEE Transactions on*, vol. 32, no. 8, pp. 608–624, 2006.

[8] M. P. Heimdahl, D. George, and R. 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.

[9] M. Staats, G. Gay, M. Whalen, and M. Heimdahl, "On the danger of coverage directed test case generation," in *Fundamental Approaches to Software Engineering*, ser. Lecture Notes in Computer Science. Springer, 2012, vol. 7212, pp. 409–424.

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

[11] J. Blom, "Model-Based Protocol Testing in an ERLANG Environment," Ph.D. dissertation, Dept. of Computer Systems, Uppsala University, Sweden, Uppsala, Sweden, 2016.

[12] 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*, ser. Lecture Notes in Computer Science, vol. 3395. Springer-Verlag, 2004, pp. 125–139.

[13] G. Rothermel, S. Elbaum, A. G. Malishevsky, P. Kallakuri, and X. Qiu, "On test suite composition and cost-effective regression testing," *ACM Trans. Softw. Eng. Methodol.*, vol. 13, no. 3, pp. 277–331, 2004.

[14] *Mobile Location Protocol*, OMA, Apr. 2004, version 1.1. [Online]. Available: http://openmobilealliance.org/

[15] E. Asarin, M. Bozga, A. Kerbrat, O. Maler, M. Pnueli, and A. Rasse, "Data structures for the verification of timed automata," in *Hybrid and Real-Time Systems*, O. Maler, Ed. Grenoble, France: Springer Verlag, LNCS 1201, 1997, pp. 346–360. [Online]. Available: citeseer.ist.psu.edu/asarin97datastructures.html

[16] *Erlang Open Telecom Platform*, Ericsson, Oct. 2015. [Online]. Available: http://erlang.org/

[17] 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.

[18] J. Tretmans and E. Brinksma, "Torx: Automated model-based testing," in *First European Conference on Model-Driven Software Engineering*, A. Hartman and K. Dussa-Ziegler, Eds., December 2003, pp. 31–43. [Online]. Available: http://doc.utwente.nl/66990/

[19] P. Santos-Neto, R. F. Resende, and C. Pádua, "An evaluation of a model-based testing method for information systems," in *Proceedings of the 2008 ACM symposium on Applied computing*, ser. SAC '08. New York, NY, USA: ACM, 2008, pp. 770–776. [Online]. Available: http://doi.acm.org/10.1145/1363686.1363865

[20] W. Grieskamp, N. Kicillof, K. Stobie, and V. Braberman, "Model-based quality assurance of protocol documentation: Tools and methodology," *Softw. Test. Verif. Reliab.*, vol. 21, no. 1, pp. 55–71, Mar. 2011. [Online]. Available: http://dx.doi.org/10.1002/stvr.427

[21] M. Sarma, P. V. R. Murthy, S. Jell, and A. Ulrich, "Model-based testing in industry: a case study with two MBT tools," in *Proceedings of the 5th Workshop on Automation of Software Test*, ser. AST '10. New York, NY, USA: ACM, 2010, pp. 87–90. [Online]. Available: http://doi.acm.org/10.1145/1808266.1808279

[22] 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*. New York, NY, USA: ACM, 2005, pp. 392–401.

[23] S. Weißleder, "Test models and coverage criteria for automatic model-based test generation with uml state machines." Ph.D. dissertation, Humboldt University of Berlin, 2010.

[24] M. Weiglhofer, B. K. Aichernig, and F. Wotawa, "Fault-based conformance testing in practice." *Int. J. Software and Informatics*, vol. 3, no. 2-3, pp. 375–411, 2009.

[25] T. Janhunen, I. Niemelä, J. Oetsch, J. Pührer, and H. Tompits, "Random vs.structure-based testing of Answer-Set programs: An experimental comparison," in *Logic Programming and Nonmonotonic Reasoning*, ser. Lecture Notes in Computer Science, J. Delgrande and W. Faber, Eds. Springer Berlin / Heidelberg, 2011, vol. 6645, pp. 242–247, 10.1007/978-3-642-20895-9_26.