RANDOM VS. SCENARIO-BASED VS. FAULT-BASED TESTING
An Industrial Evaluation of Formal Black-Box Testing Methods

Martin Weiglhofer and Franz Wotawa
Institute for Software Technology, Graz University of Technology, Inffeldgasse 16b/II, 8010 Graz, Austria

Keywords: Formal Methods, Evaluation, Input/Output Conformance (ioco), Random Testing, Test Purposes, Fault-based Testing, TGV, TorX.

Abstract: Given a formal model of a system under test there are different strategies for deriving test cases from such a model systematically. These strategies are based on different underlying testing objectives and concepts. Obviously, their usage has impact on the generated test cases. In this paper we evaluate random, scenario-based and fault-based test case generation strategies in the context of an industrial application and assess the advantages and disadvantages of these three strategies. The derived test cases are evaluated in terms of coverage and in terms of the detected errors on a commercial and on an open source implementation of the Voice-Over-IP Session Initiation Protocol.

1 INTRODUCTION

Due to the complexity of todays software systems, testing becomes more and more important. Especially, for safety-critical systems and for high availability systems software testing is essential. However, software testing is a tedious, time consuming, expensive and error prone task. Assessing the correctness of a software system with respect to a textual specification or at least getting a high confidence of the correctness requires systematic testing.

Model-based test case generation techniques claim to address these issue, by deriving test cases from a given formal model. There are different strategies for generating the test cases. The simplest used strategy is test generation based on randomness. The test generation algorithm relies on random decisions during test case synthesis. A second strategy is to guide test case generation by user specified scenarios. These scenarios tell the test generation tools for which parts of the model they should generate test cases. A third possible strategy is to use anticipated fault models in order to test for particular faults. Theoretically, this allows to prevent a system from implementing concrete faults at the specification level.

Mature research prototypes (e.g. TGV (Jard and Jéron, 2005), TORX (Tretmans and Brinksma, 2003), ...) and a sound underlying theory (Tretmans, 1996) suggest the application of such formal methods for black box testing in industrial projects. Existing case studies (Fernandez et al., 1997; Kahlouche et al., 1998, 1999; Laurencot and Salva, 2005; Philipps et al., 2003; Kovács et al., 2003), report on the application of formal testing techniques to different sized applications. Basically, they report on the application of a particular test generation technique to certain problems. In difference to that du Bousquet et al. (2000) report on comparing the two tools TORX and TGV when detecting faulty versions of an conference protocol implementation. Pretschner et al. (2005) provides a comparison of hand-crafted and automatically generated test cases (using a single strategy) in terms of error detection, model coverage and implementation coverage on an automotive network controller. In difference to that, we focus on the evaluation of different test generation strategies.

However, applying random, scenario-based and fault-based test case generation techniques in an industrial settings raises some open questions of importance: (1) Which of the three strategies work best? Does one of these strategies outperform the others? (2) What are the benefits and drawbacks of these strategies? (3) Given a formal specification what is the additional effort needed to apply these techniques to concrete industrial applications? In this paper we assess the mentioned methods in order to provide some answers to these three questions.

Of course, there are other aspects, beside the men-

Weiglhofer M. and Wotawa F. (2008).
RANDOM VS. SCENARIO-BASED VS. FAULT-BASED TESTING - An Industrial Evaluation of Formal Black-Box Testing Methods.
In *Proceedings of the Third International Conference on Evaluation of Novel Approaches to Software Engineering*, pages 115-122
DOI: 10.5220/0001764501150122
Copyright © SciTePress

tioned questions, which prevent companies from using formal methods within their software engineering processes. These aspects include educating students in writing formal specifications and increasing the usability of the available tools as well as proving their applicability to real-world systems. Even these problems have been known in the past they are still valid today and have to be tackled. However, we belief that providing proof of concepts studies and results obtained from real world examples like in this paper will help to increase the use of formal methods in practice.

This paper continues as follows: in Section 2 we briefly introduce the underlying formal theory. The used test case generation strategies and the available tools are discussed in Section 3. Section 4 presents the obtained figures and a discussion of our empirical evaluation. We draw our final conclusions in Section 5.

2 USING FORMAL METHODS FOR TEST CASE GENERATION

There are various techniques for automatic test case generation using formal models. Among others, test cases can be derived from finite state machines (FSM), i.e. the formal model is represented as a FSM. An overview of FSM based testing techniques is given by Lee and Yannakakis (1996). Other methods use model-checkers to automatically derive test cases from formal models. Fraser et al. (2007) presents a survey on state of the art model-checker based test case generation.

Another approach which has been used for the evaluation presented in this paper, relies on input-output labeled transition systems (IOLTS) for test case generation.

Figure 1 illustrates an example IOLTS, representing a vending machine that gives tea (!tea) if one coin (?1) is inserted or returns the inserted coin (!1). If it gives tea the machine cleans the internal pipes and resets itself to the initial state. This cleaning action is unobservable (τ). If a user inserts successively two coins (!1, !1) it either gets coffee or the machine returns the two inserted coins (!2). Again, after dispensing coffee the machine (invisibly) cleans its pipes in order to get ready for new orders.

In general, an IOLTS consists of a finite set of states and labeled edges connecting these states. Labels are either input-labels, denoted by ?, or output-labels, expressed by !. The label τ represents an unobservable action.

The input-output conformance testing theory of Tretmans (1996) uses a conformance relation between

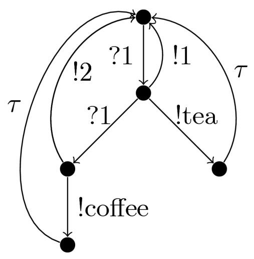

Figure 1: Example of a labeled transition system representing a vending machine.

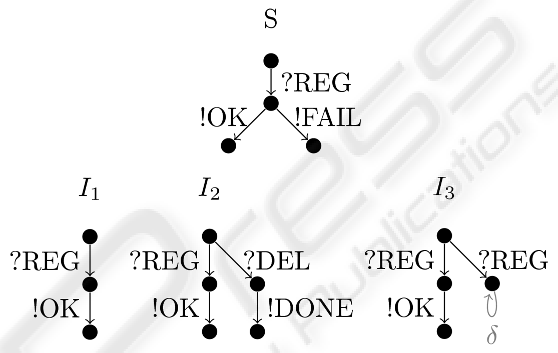

Figure 2: Specification and three different implementations of a simple registration protocol.

IOLTSs in order to express the conformance between implementations and their specifications. This theory says that an implementation under test (IUT) conforms to a specification (S), iff the outputs of the IUT are outputs of S after an arbitrary suspension trace of S. The examples of Figure 2 serve to illustrate this conformance relation.

The first implementation I_1 of Figure 2 is input-output conform to the specification S, since the definition says, that outputs of I have to be allowed by the specification, which is obviously true for any possible trace of S. Also the second implementation I_2 is an implementation of S. Since, the input-output conformance relation (ioco) only argues over all possible traces of S an implementation may behave arbitrary on traces that are unspecified in S. This reflects the practical fact, that specifications may be incomplete. I_3 non-deterministically answers with *!OK* or does not answer at all (right branch after *?REG*). In order to detect such faulty behavior, ioco introduces quiescence. A quiescent state is a state which either does not have any output (!) edges nor any internal edges (τ). Such states are marked with a special edge labeled with δ. The suspension traces used by the ioco relation are traces possibly containing δ actions. Thus, I_3 is not conforming to the specification S, because the outputs of I_3 after the trace *?REG* are $\{!OK, \delta\}$ while the specification only allows $\{!OK, !FAIL\}$.

In summary, the ioco relation seems to be applicable to industrial sized problems because: (1) it is suited to handle incomplete specifications, and (2) it allows for using non-deterministic specifications. Both issues are important because there are hardly deterministic or complete specifications available in practice. Non-determinism is typically introduced by abstracting from unnecessary real-world details within specifications. Nevertheless, the ioco relation relies on two assumptions. First, it assumes that quiescence can be detected, which is usually done by using timeouts. Second, it requires that the application accepts any input at any time, which is often true for robust protocols, but may cause some problems for other applications.

Based on this conformance relation we can generate test cases that examine whether an implementation conforms to a given specification or not.

3 THREE TECHNIQUES FOR MODEL-BASED TESTING

There are several strategies for deriving test cases from models with respect to input-output conformance testing. Among others, commonly used strategies are: (1) random testing, (2) scenario-based testing using test purposes, and (3) fault based testing. Thus, we focus on these three techniques.

3.1 TORX - Random

The TORX tool (Tretmans and Brinksma, 2003) examines an implementation under test by traversing a specification's state space randomly. In each state TORX randomly selects between sending an stimulus to the system under test (SUT) or waiting for an response from the SUT if both options are allowed by the formal specification. If only sending stimuli is allowed TORX chooses randomly one of the possible stimuli. Otherwise TORX waits for an response from the system under test.

This procedure is continued until a difference between the implementation and the specification is detected or until a certain test sequence length has been reached.

3.2 TGV - Scenario-based

The TGV tool (Jard and Jéron, 2005), which comes with the CADP toolbox (Garavel et al., 2002), uses test purposes for focusing the test generation process. A test purpose allows to cut parts of the specification which are not relevant for a particular testing scenario. By the use of a set of test purposes test generation can be focused on relevant scenarios. Thus, we have control over the generated test cases.

For a given test purpose TGV either derives one test case equipped with *Pass*, *Fail* and *Inconclusive* states or a complete test graph representing all test cases corresponding to the given test purposes.

When the testing activity ends in an *Inconclusive* state the implementation has not done anything wrong, but the system's response leads to a part of the specification that has been cut by the test purpose.

3.3 TGV - Fault-based

Aichernig and Delgado (2006) propose a technique that allows to generate test purposes based on anticipated fault models. By the use of mutation operators they generate faulty version, i.e. mutants, from the original specification. Then they construct the IOLTS S_τ for the original specification and minimize S_τ to S. This minimization removes all unobservable τ actions from S_τ. In addition, Aichernig and Delgado (2006) construct a minimized IOLTS S^M for every mutant M. An equivalence check between an mutant's IOLTS and the specification's IOLTS gives a discriminating sequence if there is an observable difference between S and S^M. This discriminating sequence is used as a test purpose which leads to a test case that fails on implementations that implement the mutant.

This procedure suffers from scalability issues, because it requires the construction of the complete state spaces for the specification and the mutant. Aichernig et al. (2007a) overcome this issue by exhibiting the knowledge of the position of the injected fault in order to search for a discriminating sequence on the relevant parts of the state spaces only.

4 COMPARING THESE THREE TECHNIQUES IN PRACTISE

We use the session initiation protocol (SIP) for the evaluation of the three discussed test case generation strategies. In order to assess the different testing strategies we executed the generated test cases against the open source implementation OpenSER and against a commercial implementation of the SIP Registrar. We conducted all our experiments on a PC with an AMD Athlon(tm) 64 X2 Dual Core Processor 4200+ and 2GB RAM.

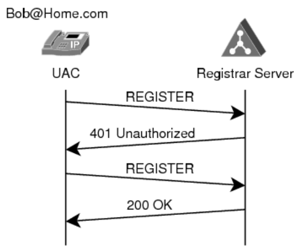

Figure 3: Simple Call-Flow of the registration process.

4.1 System under Test: Session Initiation Protocol

The Session Initiation Protocol (SIP) handles communication sessions between two end points. The focus of SIP is the signaling part of a communication session independent of the used media type between two end points. More precisely, SIP provides communication mechanisms for *user management* and for *session management*. *User management* comprises the determination of the location of the end system and the determination of the availability of the user. *Session management* includes the establishment of sessions, transfer of sessions, termination of sessions, and modification of session parameters.

SIP defines various entities that are used within a SIP network. One of these entities is the so called Registrar, which is responsible for maintaining location information of users.

An example call flow of the registration process is shown in Figure 4.1. In this example, Bob tries to register his current device as end point for his address Bob@home.com. Because the server needs authentication, it returns “401 Unauthorized”. This message contains a digest which must be used to re-send the register request. The second request is encrypted using the HTTP-Digest method described by Franks et al. (1999). This request is accepted by the Registrar and answered with “200 OK”. The full description of SIP is given by Rosenberg et al. (2002).

In cooperation with our industry partner’s domain experts we developed a formal specification covering the full functionality of a SIP Registrar. This obtained LOTOS specification comprises approx. 3KLOC (net.), 20 data types (contributing to net. 2.5KLOC), and 10 processes. Note, that the Registrar determines response messages through evaluation of the request data fields rather than using different request messages. Thus, our specification heavily uses the concept of abstract data types. Details of our SIP Registrar specification can be found in (Weiglhofer, 2006).

4.2 Benefits and Drawbacks of the Three Techniques

Table 1 gives a general overview of using the different techniques in a practical setting. This table shows for each of the three techniques (1st column) whether, given a specification, the test generation can be fully automated or if some additional manual work is necessary (2nd column). The 3rd column shows the average length of the executed test sequences. The next columns depict, the average time needed to generate a single test case (4th column) and the overall number of generated test cases (5th column). Note, that we used different numbers of test runs for random testing. While Table 1 only contains results for 5000 test runs, Section 4.3 lists detailed information on the different experiments. In addition, Table 1 shows the code coverage[1] on the open source implementation in terms of function coverage (6th column), condition/decision coverage (7th column), and decision coverage (8th column). Because of technical reasons we are not able to provide coverage measurements for the commercial implementation.

In addition, Table 1 illustrated the number of detected faults within the open source implementation (9th column) and within the commercial SIP Registrar (10th column). Finally, the 11th column illustrates whether all known faults of the implementations are found by a particular technique.

As it can be seen from that table, the average length of the executed test cases is approximately twice as long for the random testing strategy as for the other two techniques. Note, that we limited the maximum length of test sequences to 25 steps for random testing. However, the generation of fault-based test cases requires much more time than random or scenario based testing.

The code coverage shows some interesting properties of the generated test cases. First of all, random testing covers less functions than the test cases derived from our scenarios. This is because there is a complex scenario which require a particular sequence of test messages in order to put the Registrar into a certain state. If the Registrar is in this state it uses additional functions for processing REGISTER requests. Unfortunately, the random test generation never selected this sequence from the formal specification.

The condition/decision (C/D) coverage achieved by random testing is higher than the C/D coverage from scenario-based testing. That means, that random

[1]For coverage measurements we use the Bullseye Coverage Tool: http://www.bullseye.com

Table 1: Overview of the main results using random, scenario-based and fault-based test case generation techniques.

Technique	auto-mated	seq. length	test gen.time	test cases	avg. coverage			detected faults		all faults
					F	C/D	D	o.s.	comm.	
random	✓	10.95	4s	5000	73%	38%	42%	4	5	✗
scenarios	✗	4.53	1s	5408	78%	36%	40%	4	9	✗
fault-based	✓	4.78	45m33s	72	70%	30%	32%	4	6	✗

testing has inspected the covered functions more thoroughly. Our fault-based test cases achieve less code coverage than the other two approaches. However, we only need to execute 72 test cases which requires less effort for the analysis of the test results.

All applied test cases together detected 11 different faults in the commercial implementation and 5 different discrepancies between the open source implementation and our specification. The faults detected by the fault-based test cases are also detected by the scenario based approach. The random testing approach revealed in both implementations one fault that has not been detected by the other two techniques. The detected faults occur on messages sequences that have not been selected within the scenario based testing. Also the fault-based test case generation approach did not come up with this sequences.

Overall, the results illustrated by this table are disappointing, because there is no single technique which found all known faults, i.e. the faults detected by all three techniques together, on both implementations. Thus, each technique has its different strengths and weaknesses when applied in an industrial project, which will be analyzed in the following.

4.3 TorX- Random

In order to test a certain application using TorX and a given specification, a test driver needs to be implemented. The aim of a test driver is to convert abstract test messages to concrete stimuli for the system under test (SUT) and to transform responses from the SUT to abstract events that match within the specification. Because writing test drivers is a time consuming task we implemented a more generic test driver based on the rule-based rewriting system of Peischl et al. (2007).

Random Testing based on the input-output conformance relation using TorX basically has following advantages and disadvantages:

✓ Testing may result into test sequences of arbitrary length (up to a certain bound)

✓ Given a specification and a test driver, testing is fully automated

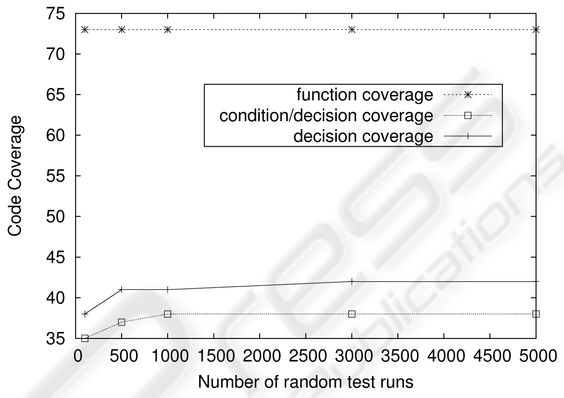

Figure 4: Code coverage on the source code when using different numbers of random test runs on the open source implementation.

✓ Since, the relevant states of the specification are constructed during test execution, TorX is applicable to huge specifications

✗ There is no possibility for guiding the testing process, e.g. in order to get good model coverage

Figure 4 illustrates the evolvement of different coverage criteria when increasing the number of random test runs on the source code of the OpenSER Registrar. This diagram shows, that the function coverage does not increase at all when increasing the number of random test runs on the open source Registrar. This is basically, because the called functions are almost always the same for most of the REGISTER message. However, the code coverage within these functions (reflected by condition/decision and by decision coverage) increases if we use more random test sequences.

4.4 TGV - Scenario-based

For our specification, we identified five relevant scenarios within the RFC (Rosenberg et al., 2002). For each scenario we wrote a single test purpose for which we generated all test cases using the algorithm of Aichernig et al. (2007b).

During test execution, we reset the implementation under test before running a certain test case in

Table 2: Test execution results of using test case derived from five manually written test purposes on the two different SIP Registrar implementations.

Test	gen.	OpenSER		Comm.	
Purpose	time	pass	fail	pass	fail
notfound	12s	880	0	0	880
invalid req.	12s	1008	320	0	1328
unauth.	15s	130	302	260	172
ok	12s	1104	384	1104	384
delete	1h57m	1148	132	16	1264
Total	1h58m	4270	1138	1380	4028

order to ensure a particular system state.

A closer look on this technique leads to following advantages and disadvantages:

✓ Using test purposes gives the test engineer more or less precise control over the generated test cases

✓ Due to an incremental test case generation TGV does not require the complete specification's state space, which makes this technique applicable to large sized (infinite) specifications.

✗ Test purposes allow to specify refuse states, which cut the search space and consequently decrese the test generation time. If test purposes lack of such states test generation may become slow.

✗ Given a specification, some additional time is needed to develop a set of test purposes

✗ To our best knowledge, there are currently no techniques available to evaluate the quality of test purposes

✗ Depending on the test purposes this procedure may lead to many test cases possibly failing because of the same root causes.

The results of executing the derived test cases against the two different implementations are illustrated in Table 2. This table shows the test generation time in seconds (2nd column), for each test purpose (1st column). In addition, this table contains the number of passed (3rd and 5th column) and the number of failed test cases (4th and 6th column) for the open source implementation (3rd and 4th column) and the commercial implementation (5th and 6th column).

As there can be seen, we have many failed test cases. Analyzing this failed test cases leads to 9 different faults on the commercial implementation and 4 different faults on the open source Registrar. Thus, many test cases fail because of the same errors, which makes test result analysis a time consuming task.

Table 3: Test execution results of using fault-based test cases on the two different SIP Registrar implementations.

Mutation	OpenSER			Commercial		
Operator	pass	fail	inc.	pass	fail	inc.
EIO	0	7	18	3	20	2
EIO+	1	7	27	9	23	3
ESO	0	4	1	0	4	1
MCO	0	5	2	0	2	5
EIO/a	8	0	17	6	0	19
EIO+/a	8	0	27	8	0	27
ESO/a	2	1	2	3	1	1
MCO/a	0	5	2	0	5	2
Total	19	29	96	29	55	60

4.5 TGV - Fault-based

We developed a mutation tool that takes a LOTOS specification and uses some mutation operators (association shift operator [ASO], event drop operator [EDO], event insert operator [EIO], event swap operator [ESO], missing condition operator [MCO]) of Black et al. (2000) and of Srivatanakul et al. (2003) in order to generate faulty versions (mutants) of the specification for each possible mutation. In addition, our mutation tool inserts markers such that we are able to extract the specification's relevant part only (see Section 3.3).

By the use of this mutation tool we generated 95 faulty versions of our specification. 23 of this 95 mutants do not exhibit an observable fault, thus we get 72 test purposes using the fault based test purpose generation technique. For a fault-based test purpose it is sufficient to use a single test case derived from this test purpose, since every test case of the test purpose will fail if the faulty specification has been implemented.

The average time needed to derive a test case from a faulty version of the specification is approximately 40 minutes. This average is high because of some complex mutants. Anyway, for the majority of mutants (94%) the corresponding test case is generated within 16 minutes.

All fault-based generated test cases follow the same structure. They start with some preamble which brings the implementation to a particular state that possibly exhibits the faulty behavior. There they try to observe whether or not the faulty specification has been implemented.

Table 3 show the obtained results when executing the derived test cases on our two implementations of the SIP Registrar. This table lists for each mutation operator (1st column), the number of passed (2nd and 5th column), failed (3rd and 6th column) and inconclusive (4th and 7th column) test cases for the open source (2nd, 3rd and 4th column) and the commercial Registrar (5th, 6th and 7th column).

Note, that we executed the derived test cases against the implementations using two different configurations. The results from the 2nd to the 5th row show the results when authentication was turned off, while the results from the 6th to the 9th row show the results for the Registrars when authentication was turned on.

The used EIO+ operator is a derivative of the EIO operator but inserts an output event instead of inserting an event by duplicating a previous event. This generates more observable faults, since the EIO operator may duplicate τ events which results into an unobservable difference between the mutant and the specification.

We identified following advantages and disadvantages of the fault-based approach:

✓ The fault-based test case generation strategy allows to test for the absence of particular faults at the level of the specification.

✓ Given a formal specification the test generation can be fully automated.

✓ We need only one test case per mutant which results into manageable test suite sizes.

✗ The overall test case generation process, especially the test purpose generation, is a time consuming task.

✗ If the generated test cases fail to execute their preambles, testing for a specific fault may be infeasible, i.e., test cases terminate with inconclusive verdicts.

5 CONCLUSIONS

In this paper we have evaluated random, scenario-based and fault-based test generation techniques, by testing two implementations of an Session Initiation Protocol Registrar. A closer look on the empirical results allows to answer the initially stated questions.

Which of the Three Strategies Work Best? Our evaluation shows that no one of these three methods outperforms the others. The fault-based approach works worst and reveals no additional failures. The random and the scenario-based approach perform almost equal and detect failures that are not found by the other approaches.

What are the Benefits and Drawbacks. Basically, all of the evaluated testing strategies allow to systematically derive test cases for industrial-sized specifications. The quality and the number of the derived test cases differ. Fault-based test generation gives a low number of test cases with a code coverage lower than that of the other two techniques. Random testing may require a long time to find particular failures, but possibly finds failures overseen by scenario-based testing. Scenario-based testing uses user specified scenarios to guide the test generation process. In that case, the overall quality of the test cases depend on the quality of the specified scenarios.

What are the Additional Efforts. The model-based testing techniques start from a formal model. Typically, the industry refuses from writing such models since they belief that this is a time consuming. Reasons for that are the complexity of nowadays formal modelling languages and the lack of good tool support. As concluded by Pretschner et al. (2005), we also belief that the use of models pay off in terms of failure detection. In addition, to the formal model a test driver which converts test messages to concrete system inputs is needed in order to execute the derived test cases. However, given a formal specification and a test driver, fault-based testing and random testing can be completely automated. For scenario based testing the user needs to specify test purposes manually. These test purposes allow to control the test generation process, but require additional work.

Finally, we have to mention that we were able to reveal several differences between the specification and its implementations. Since the implementations are deployed in industrial Voice-over-IP networks this case study shows the relevance of model-based testing for industry.

ACKNOWLEDGEMENTS

The research herein is partially conducted within the competence network Softnet Austria (www.soft-net.at) and funded by the Austrian Federal Ministry of Economics (bm:wa), the province of Styria, the Steirische Wirtschaftsförderungsgesellschaft mbH. (SFG), and the city of Vienna in terms of the center for innovation and technology (ZIT).

We are grateful to Alexander Pilz for his help with the implementation of the TORX adapter module.

REFERENCES

Aichernig, B. K. and Delgado, C. C. (2006). From faults via test purposes to test cases: On the fault-based testing of concurrent systems. In *Proceedings of the 9th International Conference on Fundamental Approaches to Software Engineering*, volume 3922 of *LNCS*, pages 324–338. Springer.

Aichernig, B. K., Peischl, B., Weiglhofer, M., and Wotawa, F. (2007a). Protocol conformance testing a SIP registrar: An industrial application of formal methods. In Hinchey, M. and Margaria, T., editors, *Proceedings of the 5th IEEE International Conference on Software Engineering and Formal Methods*, pages 215–224, London, UK. IEEE.

Aichernig, B. K., Peischl, B., Weiglhofer, M., and Wotawa, F. (2007b). Test purpose generation in an industrial application. In *Proceedings of the 3rd International Workshop on Advances in Model-Based Testing*, pages 115–125, London, UK.

Black, P. E., Okun, V., and Yesha, Y. (2000). Mutation operators for specifications. In *Proceedings of the 15th IEEE International Conference on Automated Software Engineering*, pages 81–88, Grenoble, France. IEEE.

du Bousquet, L., Ramangalahy, S., Simon, S., Viho, C., Belinfante, A., and de Vries, R. G. (2000). Formal test automation: The conference protocol with TGV/-TORX. In *Proceedings of 13th International Conference on Testing Communicating Systems: Tools and Techniques*, volume 176 of *IFIP Conference Proceedings*, pages 221–228, Dordrecht. Kluwer Academic Publishers.

Fernandez, J.-C., Jard, C., Jéron, T., and Viho, C. (1997). An experiment in automatic generation of test suites for protocols with verification technology. *Science of Computer Programming*, 29(1-2):123–146.

Franks, J., Hallam-Baker, P., Hostetler, J., Lawrence, S., Leach, P., Luotonen, A., and Stewart, L. (1999). HTTP authentication: Basic and digest access authentication. RCF 2617, IETF.

Fraser, G., Wotawa, F., and Ammann, P. (2007). Testing with model checkers: A survey. Technical Report SNA-TR-2007-P2-04, Competence Network Softnet Austria.

Garavel, H., Lang, F., and Mateescu, R. (2002). An overview of CADP 2001. *European Association for Software Science and Technology Newsletter*, 4:13–24.

Jard, C. and Jéron, T. (2005). TGV: theory, principles and algorithms. *International Journal on Software Tools for Technology Transfer*, 7(4):297–315.

Kahlouche, H., Viho, C., and Zendri, M. (1998). An industrial experiment in automatic generation of executable test suites for a cache coherency protocol. In *11th International Workshop on Testing Communicating Systems*, IFIP Conference Proceedings, pages 211–226. Kluwer.

Kahlouche, H., Viho, C., and Zendri, M. (1999). Hardware testing using a communication protocol conformance testing tool. In *Proceedings of the 5th International Conference Tools and Algorithms for Construction and Analysis of Systems*, volume 1579 of *LNCS*, pages 315–329. Springer.

Kovács, G., Pap, Z., Viet, D. L., Wu-Hen-Chang, A., and Csopaki, G. (2003). Applying mutation analysis to sdl specifications. In *Proceedings of the 11th International SDL Forum*, LNCS, pages 269–284, Stuttgart, Germany. Springer.

Laurencot, P. and Salva, S. (2005). Testing mobile and distributed systems: Method and experimentation. In Higashino, T., editor, *Proceedings of the 8th International Conference on Principles of Distributed Systems*, volume 3544 of *LNCS*, pages 37–51. Springer.

Lee, D. and Yannakakis, M. (1996). Principles and methods of testing finite state machines - a survey. *Proceedings of the IEEE*, 84(8):1090–1123.

Peischl, B., Weiglhofer, M., and Wotawa, F. (2007). Executing abstract test cases. In *Model-based Testing Workshop in conjunction with the 37th Annual Congress of the Gesellschaft fuer Informatik*, pages 421–426, Bremen, Germany. GI.

Philipps, J., Pretschner, A., Slotosch, O., Aiglstorfer, E., Kriebel, S., and Scholl, K. (2003). Model-based test case generation for smart cards. *Electronic Notes in Theoretical Computer Science*, 80:1–15.

Pretschner, A., Prenninger, W., Wagner, S., Kühnel, C., Baumgartner, M., Sostawa, B., Zölch, R., and Stauner, T. (2005). One evaluation of model-based testing and its automation. In *Proceedings of the 27th International Conference on Software Engineering*, pages 392 – 401, St. Louis, Missouri, USA. ACM.

Rosenberg, J., Schulzrinne, H., Camarillo, G., Johnston, A., Peterson, J., Sparks, R., Handley, M., and Schooler, E. (2002). SIP: Session initiation protocol. RFC 3261, IETF.

Srivatanakul, T., Clark, J. A., Stepney, S., and Polack, F. (2003). Challenging formal specifications by mutation: a csp security example. In *Proceedings of the 10th Asia-Pacific Software Engineering Conference*, pages 340–350. IEEE.

Tretmans, J. (1996). Test generation with inputs, outputs and repetitive quiescence. *Software - Concepts and Tools*, 17(3):103–120.

Tretmans, J. and Brinksma, E. (2003). TorX: Automated model based testing. In Hartman, A. and Dussa-Zieger, K., editors, *Proceedings of the 1st European Conference on Model-Driven Software Engineering*, pages 13–25, Nurnburg, Germany.

Weiglhofer, M. (2006). A LOTOS formalization of SIP. Technical Report SNA-TR-2006-1P1, Competence Network Softnet Austria, Graz, Austria.