Project no.:	**ICT-FP7-STREP-214755**
Project full title:	**Quantitative System Properties in Model-Driven Design**
Project Acronym:	**QUASIMODO**
Deliverable no.:	**D4.3**
Title of Deliverable:	**Test selection and coverage**

Contractual Date of Delivery to the CEC:	**Month 30**
Actual Date of Delivery to the CEC:	**June 1, 2011**
Organisation name of lead contractor for this deliverable:	**ESI/UT**
Author(s):	**Henrik Bohnenkamp** **Mark Timmer, Arne Skou**
Participants(s):	**P01 AAU, P02 ESI/UT, P04 RWTH**
Work package contributing to the deliverable:	**WP 4**
Nature:	**R**
Version:	**1.0**
Total number of pages:	**18**
Start date of project:	1 Jan. 2008 **Duration:** 36 month

Project co-funded by the European Commission within the Seventh Framework Programme (2007-2013)
Dissemination Level
PU Public

Abstract:

This deliverable reports on the results in the area of test selection and coverage for quantitive system, produced in the QUASIMODO project.

Keyword list: test selection, coverage.

Contents

Abbreviations

AAU: Aalborg University, DK (P01)

ESI: Embedded Systems Institute, NL (P02)

ESI/UT: University of Twente, Enschede, NL (under the auspices of P02)

RWTH: RWTH Aachen University, D (P04)

1 Introduction

Software becomes more and more complex, making thorough testing an indispensable part of the development process. The U.S. National Institute of Standards and Technology has assessed that software faults cost the American economy almost sixty billion dollars annually [18]. More than a third of these costs could be eliminated if testing occurred earlier in the development process.

An important fact about testing is that it is inherently incomplete, since testing everything would require infinitely many input scenarios. On the other hand, passing a well-designed test suite does increase the confidence in the correctness of the tested product. Therefore, it is important to select test cases in the most efficient and effective way beforehand, and to assess the quality of a test process after it has taken place.

In industry, several methods are in use to select which test cases to include in a test suite. Well-known are syntactic coverage measures such a statement coverage and condition coverage [16]. They base the quality of a test suite on the percentage of statements or conditions of the implementation that are executed during testing, and steer the construction of a test suite in the direction that obtains a coverage measure of 100%.

A large amount of the current research on testing focusses on the area of *model-based testing* (MBT): using a model of the specification to automatically generate, execute and evaluate test cases [11]. This technique makes it possible to perform many more tests than would ever be possible manually. However, MBT, and in particular test selection when applying MBT, still faces several problems both in theory and in practice. The following issues are addressed in the work described by this document and performed in the context of the Quasimodo project.

Under what circumstances should a system fail or pass a test case? When designing and selecting test cases, it should be decided when the system is considered to respond correctly to that test case. For this purpose, conformance relations have been introduced. In the context of model-based testing an important conformance relation is *ioco* [20] (input-output conformance). It allows systems to handle unexpected input, but does not allow unexpected outputs (neither does it allow the unexpected absence of outputs, called *quiescence*).

Conformance relations were also already defined for timed systems [14] and for symbolic systems [12], but not yet for a combination of both. This gap has now been filled by [4], where a conformance relations for Symbolic Timed Automata (a combination of Timed Automata and Symbolic Transition Systems) is introduced. This work is discussed in Section 2.

How to derive timed test cases for a complex system? The bigger the system, the more difficult it is to design an effective and efficient test suite. Ideally, these tests are generated automatically from a model, executed against an implementation-under-test and evaluated according to some conformance relation. However, it is not always easy to obtain a test, especially not in a setting with real time.

In Section 3 we discuss a UPPAAL-based tool for deriving test cases based on timed automata [2]. It allows the user to make a model in UPPAAL, and then generate a test suite with complete edge coverage based on several test generation algorithms. Either the test suite is generated based on a reachability question or an optimization strategy, by targeting single edges, or just randomly (to support these techniques, two model transformations are performed). To generate an actual executable test suite, the user can annotate the model to denote in what way inputs have to be provided and observations have to be made.

How to interpret a successful testing process? Although it is impossible to exhaustively test any non-trivial system, passing a well-designed test suite does increase the confidence in the correctness of the tested product. Therefore, it is important to assess the quality of a test suite. In the past, two fundamental concepts have been put forward to evaluate test suite quality: (1) *coverage metrics* determine which portion of the requirements and/or implementation-under-test has been exercised by the test suite; (2) *risk-based metrics* assess the risk of putting the tested product into operation. There are still, however, major issues regarding the usefulness of these concepts. A new, quantitative approach to deal with these issues has been proposed in [3] and is discussed in Section 4.

Still, the traditional coverage notions are also still used quite extensively in practice. Companies assume that test suites with for instance a high percentage of statement coverage also perform well in the sense of finding most of the existing faults. The question, however, is whether this assumption is indeed a valid one. Section 5 discusses an applied research project that investigated this question in the context of a Dutch software company.

2 A Conformance Testing Relation for Symbolic Timed Automata

Participants

- Henrik Bohnenkamp, RWTH Aachen University, D;
- Sabrina von Styp, RWTH Aachen University, D;
- Julien Schmaltz, Radboud University, NL.

Challenge

The well-known *ioco testing framework* [20] has been recently extended in two directions. First of all, *Timed Automata* have been proposed as specification formalisms in several approaches for testing real-time behaviours [14, 8, 10]. Different notions of conformance have been defined on the basis of timed LTSs ($TLTS$s), i.e., only on the semantic level. Second of all, *Symbolic Transition Systems* (STSs) [12, 13] have been introduced to specify systems with input- and output-data. STSs are LTSs extended with a notion of data and data-dependent control flow based on first-order logic. The symbolic representation of data in STSs allows for infinite data domains without facing the problems of infinite branching and infinite state spaces. For STSs, the implementation relation *sioco* has been developed, which is defined solely within the first-order logic framework on STS level [13].

In this work we develop a combination of real-time and data. In particular, in [4] we take first steps in the direction of specification-based testing for systems combining input/output data with real-time aspects in a non-orthogonal way. This means that the input data can influence the real-time behaviour. We introduce a conformance relation which takes data and real-time into account.

Results

Our contribution is a new formalism – called Symbolic Timed Automata (STAs) – for modelling reactive real-time systems with data input and output. STAs are a straightforward combination of STSs and TAs.

We defined a concrete operational semantics in terms of timed labelled transition systems, and a symbolic trace semantics for this formalism. We proved that both semantics coincide. The trace semantics is based on symbolic execution of the STAs. In the symbolic execution, time delays are also treated symbolically as data. The semantics provides first-order formulae which describe the conditions under which certain states in an STA can be reached at what times.

Based on this information, a family of conformance relations $\textit{stioco}_{\mathcal{F}_s}$ is defined, which expresses conformance of an input-enabled implementation, given as an STA, with a specification, also given as an STA. We show that $\textit{stioco}_{\mathcal{F}_s}$ coincides on the concrete semantical level with the notion of *tioco* of Krichen and Tripakis [14].

Perspective

The interaction between time and data is restricted to the influence that data inputs can have on the timing behaviour of the considered STA. This was expressed by allowing location variables to serve as bounds in clock constraints and invariants. More and different interactions between time and data are imaginable, for example, by assigning clock valuations to location variables, i.e., by keeping historical information about the occurrence time of events in the STA. In principle, this extension could also be encoded in the first-order logical framework. However, even for the more restricted case considered in this paper, it is necessary to investigate first whether the obtained formalism is not already too expressive to be useful for practical testing, in terms of decidability of the forward reachability problem. A suitable subclass of first-order logic might have to be identified to ensure this and to be able to apply the provided theory on practical applications. This would encompass the development of an algorithm for automatic test-case generation and test-execution.

3 Model-based testing with UPPAAL

Participants

- Kim Guldstrand Larsen, Aalborg University, DK;
- Jacob Illum Rasmussen, Aalborg University, DK;
- Arne Skou, Aalborg University, DK.

Challenge

This documents describes features and functionality of the UPPAAL-based testing tool for model-based testing [2]. The aim of the tool is to take models created in UPPAAL and create a suite of test cases that covers all transitions in the model (edge coverage). By using a special syntax within UPPAAL, the test suite that is the output of the tool can take the format of a test script in any desired language that can be used as input to test executions engines such as Selenium (`http://seleniumhq.org`).

Results

The test generation algorithm is a combination of the normal UPPAAL search algorithm, the random depth-first search algorithm, and the agent-based version of the UPPAAL CORA search engine.

Below, we describe in turn the modeling features, the test generation algorithms and the model transformation for test generation.

Modeling Features

Test scripts consist of a sequence of inputs to the system under test (SUT) interleaved with a sequence of observations on the SUT as a results of the input stimuli. Inputs are given in terms of synchronization channels in UPPAAL. In order to distinguish between channels indicating inputs and other channels used internally in the model of the SUT, the test tools requires the existence of a UPPAAL template called “User” (the name is case insensitive). Any channels used as calling synchronizations (i.e., with an exclamation point) in the user template are interpreted as test inputs to the SUT. An advantage of having a user template is that the user behavior can be a complex model allowing, e.g., certain inputs only under certain circumstances. Obviously, the channels used for inputs must be declared globally.

The least restrictive user model consists of a single state with a loop edge for each input channel sending that input.

To have test scripts as an output of the test generation procedure, the model must contain information about inputs and observations, and what the syntax of these is in the desired output language. This is not a native feature of UPPAAL, but to allow the user to use the regular flavor of UPPAAL for test generation, the feature has been added by interpreting UPPAAL comments in a special way. How this is done for inputs and observations is described below.

Inputs: If there is a channel used for input called for instance '`key_press_x`', then a comment can be placed in either the global declaration or in the local declaration of the user template, with the following syntax: `key_press_x => <some code here>`. Then, the text replaced for `<some code here>` will appear in the output test script whenever the test dictates that the input '`key_press_x`' should be given.

Observations: Observations come in two flavors: variables and locations. Each of these is handled differently:

Variables: These are handled in a similar fashion to inputs, that is, using the syntax `<variable name> => <some code here>`. However, there is a difference in that variables, unlike inputs, takes a certain value. So, in the code associated with the variable, the special syntax '`$val`' can be used, and will be substituted for the current value of the variable when the observation should be made. Note that variable observations occur only in the test script whenever the variable changes value and not after each single input. An example of a variable definition could be something like `foo => check_var('foo',$val);`

Locations: Whenever one of the templates in the SUT model changes location, any code inserted in the comment field of that location is inserted into the test script. However, since templates can be instantiated and thus occur in the SUT model with several different names, the special syntax '`$instance`' can be used within the code and will be replaced by the actual name used for the instance of the template that has changed location. Again, only locations that change as a result of the previous input occur in the test script as an observation.

Any valid UPPAAL model using the above features can be used for automatic test generation using the test tool.

Test Generation Algorithms

The test tool allows the user to utilize a number of different algorithms for test generation. Each of these techniques is described below:

Query-based: The query-based technique is not per se a test generation algorithm. However, it is nonetheless very useful for test case generation. This technique assumes that the user provides as input a UPPAAL query file with a number of reachability queries. Then, for each query, the test tool will generate a test case and these will be part of the final test suite. This can be useful in at least the following situations:

- When the user knows that reaching a certain state will cover a great part of the model, and thus help the tool limit the size of the final test suite.

- When the test specification of the SUT specifies certain functional tests that must exist. Often these can be translated into reachability queries and can then be (1) automatically generated and (2) count towards the coverage of the model so that the automatic tests do not cover transitions that have already been covered by the functional tests.

Optimization: Optimization means generating test cases that individually cover as much of the uncovered part of the model as possible. This is done by using the UPPAAL CORA optimization engine. The model is then converted to a CORA model and special optimization queries are executed. The decision of how much to cover with a single test case in this fashion has to be weighed against the length of the test case. Often, longer test cases are less useful, as debugging an error from a longer test case is harder than from a shorter test case. The CORA engine used is the agent-based version, which does not guarantee optimality. This is needed, as generating a test case of a certain length that covers as much of the model as possible can be an extremely hard problem to solve. Even much more complex than optimization queries on the input model, since the annotations added to the model in order to track coverage can add an explosion in the global state space. The optimization algorithm is executed iteratively until it covers no new edges.

Random Depth: Random depth involves generating traces using the built-in random depth-first search algorithm of UPPAAL. This technique is mostly useful in situations where optimization queries are not performed and before single step test cases are generated, in order to reduce the size of the final test suite. This technique requires no model transformation. The random depth algorithm is executed iteratively until no new edges are covered.

Single Step: Single step is the 'garbage collector' of the test generation algorithms as it picks up the pieces left behind by the other techniques. This technique is an iterative approach that selects a random uncovered edge in the model and generates a reachability query designed to cover that specific edge. If the query succeeds, the trace is added as a test case to the test suite. If not, the edge is known not to be reachable and is added to the list of uncoverable edges. Thus, single step is used as the final algorithm when multiple algorithms are used in order to guarantee that the final test suite covers all reachable edges in the model. Since single step chooses one edge at a time, this can result in a large number of test cases (worst case one test for each edge).

This concludes the description of algorithms used by the testing tool. All the algorithms perform edge coverage of the model, i.e., covering each edge in each template. Note that this means that two different instances of the same template share coverage of the template and do not define two individual sets of edges to cover.

Model Transformations

In order to use the input model for test generation, certain model transformation techniques need to be applied depending on the test generation algorithm. The different transformations will be described in more detail below.

Using either random depth or query-based test generation requires no model transformations, as neither needs to know which edges have already been covered. The model transformations needed for single step and optimizations are as follows.

Single Step: To perform single step test case generation, an extra boolean variable is added to the model with an initial value of 'false'. Then, an assignment setting that variable to 'true' is added to the edge that needs to be covered. Finally, a reachability query is constructed that checks when a state is reachable for which the boolean variable has value 'true'.

Optimization: Optimization requires models that involve costs, and the coverage to be obtained by any single test case should be maximal. Therefore, the model transformations for optimization are the most elaborate. First, we need to assign costs in the model. The cost scheme used by the tool is to assign a cost of 1 to each covered edge and 0 to each uncovered edge. We also add a new template, which has two locations — an initial non-goal location and the goal location. The transitions from the initial location to the goal location have a cost of X times the number of uncovered transitions. Thus, X functions as a weight between the length of a test case and the value of covering an extra edge. That is, in the model it is cheaper to traverse $X - 1$ covered edges and then traverse an uncovered edge before going to the goal location, than to more directly go to the goal location.

Moreover, we need to annotate the model in order to track the covered edges. This is done by adding an array of boolean variables, one variable for each edge. The array is initialized according to the currently covered and uncovered edges. Then, for each edge, a so-called trap formula is added, which sets the corresponding index of the array to 'true' when the edge is traversed. This is called a trap formula, since the value is "trapped" after traversing an edge as it can never be unset. With these additions we pose the reachability query whether the system can reach the goal locations. Then, the optimization engine of CORA will try to find the cheapest way of reaching the goal location, which should preferably cover a significant number of previously uncovered edges.

All of the above cover the most of the inner workings of the testing tool. The tool works by selecting the different algorithms that should be applied, which will be executed in turn using the following order: query-based, optimization, random depth, single step. This order is chosen as it is the most likely to produce an efficient test suite.

Time

When a UPPAAL model contains clocks and the resulting traces are timed, this is represented in the resulting test case as if there was a variable called '`delay`'. This means that similarly to using the translation from variable to scripting language, the same constructs can be used for the translation of delays. Also, the given value of the delay can be accessed using the '`$val`' notation. This all means that the resulting test cases will be interleavings of observations and either actions or delay.

Extra Features

For each test case that the tool generates it creates, with the test script, a UPPAAL model that shows the coverage of that test model in relation to all prior test cases. The coverage of the model is given in terms of a coloring scheme of the transitions. Blue transitions indicate that this test case traverses the given edge, but that a test case before this one already covered that edge. Green transitions indicate that the test case is the first to traverse that edge and is thus responsible for covering that edge. Red edges indicate edges not covered by this test case, but by a previous test case. Black edges are uncovered edges.

Besides indicating the coverage with colors, the model generated with the case also adds to the user template the sequence of inputs that resulted in this test case. In case the input model is deterministic, running the model in the simulator will result in showing the execution of the test case.

Perspective

Future work includes adding more coverage criteria to the tool. Such criteria would include location coverage (statement), which is a subset of edge coverage, and 2-switch coverage, which is a superset of edge coverage.

4 Interpreting a Successful Testing Process: Risk and Actual Coverage

Participants

- Marielle Stoelinga, University of Twente, NL;
- Mark Timmer, University of Twente, NL.

Challenge

Existing coverage measures such as code coverage in white-box testing [7, 16] and state and/or transition coverage in black-box testing [15, 17, 21] give an indication of the quality of a test suite, but it is not necessarily true that higher coverage implies that more, or more severe, faults are detected. This is because these metrics do not take into account where in the system faults are most likely to occur. Risk-based testing methods do aim at reducing the expected number of faults, or their severity. However, these are often informal [19], based on heuristics [6], or indicate which components should be tested best [5], but rarely quantify the risk after a successful testing process in a precise way.

A first attempt at a notion of coverage that does take into account the severity of errors was provided by Brandán Briones, Brinksma, and Stoelinga [9]. They introduced a concept we would call *potential coverage*, as it considers which faults *can* be detected during testing. This measure, however, does not yet take into account the faults that *are* actually covered during a test execution, making them not that precise.

Results

In [3], we present a framework in which risk and coverage can be defined, computed and optimised in a black-box manner, for systems with nondeterminism. Key properties are a rigorous mathematical treatment based on solid probabilistic models, and the result that lower risk (or higher coverage) implies a lower expected number of faults.

The starting point in our theory is a *weighted fault specification (WFS)*, consisting of (1) a specification describing the desired system behaviour as an input-output labelled transition system (IOLTS), (2) a weight function describing the severity of faults, (3) an error function describing the probability that a certain error has been made, and (4) a failure function describing the probability that incorrectly implemented behaviour yields a failure. From the WFS we derive its underlying probability model, i.e., a random variable that describes the distribution of (possibly erroneous) implementations. This allows us to define risk and actual coverage in an easy and precise way.

Given a WFS, we define the *risk* of a test suite as the expected fault weight that remains after this test suite passes. We show how to construct a test suite of a certain size with minimal expected risk. We also introduce *actual coverage* for a test suite, which quantifies the risk reduction obtained when an implementation passes it. Whereas risk is based on faults contained within the entire system, actual coverage only relates to the part of the system that has been tested. This matches with the traditional interpretation of coverage.

Perspective

Although our work provides a thorough mathematical foundation to be built upon, its theoretical nature makes it quite hard to apply it in practice. Some simplifications and approximations might be useful to make the framework more practical. Moreover, it would be very interesting to perform industrial case studies to assess the extent to which our notions of coverage and risk indeed predict that little faults remain. The case studies could also be used to investigate potential optimisations or simplifications to the framework.

5 Industrial validation of test coverage quality

Participants

- Martijn Adolfsen, University of Twente, NL;
- Marielle Stoelinga, University of Twente, NL;
- Mark Timmer, University of Twente, NL.

Challenge

Several testing techniques were developed to reduce the number of software faults, from formal methods to ad-hoc techniques. One of the concepts to improve software quality is using one or more coverage metrics, e.g., statement coverage. A coverage metric specifies which parts of program code should be executed, e.g., all statements. Each coverage metric has different criteria and therefore has specific requirements that a test suite should meet. When a test suite meets the requirements of a coverage metric, testers assume it is able to find more faults than a random test suite that does not meet the requirements. Based on this assumption, coverage metrics are used to quantify the quality of a test suite, i.e., the higher the percentage of coverage of test suite has, the higher its quality is perceived, and the more faults it presumably will find.

We conducted an experiment to investigate if this assumption is indeed valid. The metrics investigated are statement, branch and basis path coverage. By executing several test suites that achieve different percentages of these metrics, and by using coverage tooling, we reveal the actual quality of the test suites. Based on this, our aim is to conclude to what extent the use of coverage metrics improves test suite quality. The research was done in the context of a Dutch company called Info Support.

Results

The results of this research are presented in [1]. There appeared to be a normal to strong correlation between the coverage percentages of several metrics and software quality, which suggests that increasing the coverage percentage will likely increase the number of faults that are found by a test suite. This holds for each tested metric and coverage percentage, i.e., independent of the used metric and/or already achieved coverage percentage. However, the exact correlation remains uncertain, as practical limitations resulted in the end in only a limited number of measurements.

We observed that the different metrics performed equally, i.e., find almost the same amount of faults. What should be taken into account on this result is that (1) tested coverage percentages differed among the metrics, and (2) the cyclomatic complexity was relatively low, resulting in a reduced difference among metrics.

Interestingly, even in best-case scenarios, most metrics were able to find as little as half of the total faults. Therefore, coverage testing should definitely not be the only method in a test process. After investigating the faults that were (not) found, we can conclude that faults that manipulated the behavior of the program under normal conditions were found mostly. Faults that only occurred in more exceptional situations were mostly left undetected. It is likely that boundary-value analysis and equivalence partitioning [16] would have exposed most of these undetected faults in our case studies.

Perspective

Based on our results there are several interesting directions for future work. First, we tested very low coverage percentages for the basis path coverage metric. However, despite the low coverage percentages, test suites that did achieve these percentages covered a relatively high amount of faults. Future research may investigate the increase in fault percentage for higher coverage percentages.

Second, test cases used in this research were created based on the test-driven development methodology. Obviously, this has a direct consequence on the way the system is tested, and therefore the number of faults that are found. Further research could specifically focus on the influence that the type of test cases has on the efficiency of coverage metrics.

Third, as we mentioned in our results, most faults that were not found by the coverage metrics, could have been exposed using boundary-value analysis and/or equivalence partitioning. Therefore we suggest investigating their efficiency using industrial case studies. Finally, similar studies are needed to obtain an increased confidence on the correlation between coverage and software quality, and to validate our result. A recommendation is to increase the number of injected faults compared to this experiment. This will improve the exactness of the correlation coefficients. Also we recommend to add case studies that have a relative high cyclomatic complexity, to investigate the effect this has on the efficiency of the metrics and the costs.

References to Quasimodo Contributions

[1] M. Adolfsen. Industrial validation of test coverage quality. Master's thesis, University of Twente, 2011.

[2] U.H. Hjort, J.I. Rasmussen, K.G. Larsen, M.A. Petersen, and A. Skou. Model-based GUI testing using uppaal at Novo Nordisk. In *Proceedings of the 2nd World Congress on Formal Methods (FM '09)*, volume 5850 of *Lecture Notes in Computer Science*, pages 814–818. Springer, 2009.

[3] M.I.A. Stoelinga and M. Timmer. Interpreting a successful testing process: Risk and actual coverage. In *Proceedings of the 3rd IEEE International Symposium on Theoretical Aspects of Software Engineering (TASE '09)*, pages 251–258. IEEE Computer Society, 2009.

[4] S. von Styp, H.C. Bohnenkamp, and J. Schmaltz. A conformance testing relation for symbolic timed automata. In *Proceedings of the 8th International Conference on Formal Modelling and Analysis of Timed Systems (FORMATS '10)*, Lecture Notes in Computer Science. Springer, 2010. To appear.

References to Related and General Literature

[5] S. Amland. Risk-based testing: risk analysis fundamentals and metrics for software testing including a financial application case study. *Journal of Systems and Software*, 53(3):287–295, 2000.

[6] J. Bach. Heuristic risk-based testing. *Software Testing and Quality Engineering Magazine*, November/December 1999.

[7] T. Ball. A Theory of Predicate-Complete Test Coverage and Generation. In *Proceedings of the 3rd International Symposium on Formal Methods for Components and Objects (FMCO '04)*, volume 3657 of *Lecture Notes in Computer Science*, pages 1–22. Springer, 2004.

[8] H.C. Bohnenkamp and A. Belinfante. Timed testing with TorX. In *Proceedings of the International Symposium of Formal Methods Europe (FM '05)*, volume 3582 of *Lecture Notes in Computer Science*, pages 173–188. Springer, 2005.

[9] L. Brandán Briones, E. Brinksma, and M. I. A. Stoelinga. A semantic framework for test coverage. In *Proceedings of the 4th International Symposium on Automated Technology for Verification and Analysis (ATVA '06)*, volume 4218 of *Lecture Notes in Computer Science*, pages 399–414. Springer, 2006.

[10] L. Brandán Briones and H. Brinksma. A test generation framework for quiescent real-time systems. In *Proceedings of the 4th International Workshop on Formal Approaches to Testing of Software (FATES '04)*, volume 3395 of *Lecture Notes in Computer Science*, pages 64–78. Springer, 2005.

[11] I.K. El-Far and J.A. Whittaker. Model-based software testing. In *Encyclopedia of Software Engineering (2nd edn)*, volume 1, pages 825–837. Wiley, 2002.

[12] L. Frantzen, J. Tretmans, and T.A.C. Willemse. Test generation based on symbolic specifications. In *Proceedings of the 4th International Workshop on Formal Approaches to Software Testing (FATES '04)*, volume 3395 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2005.

[13] L. Frantzen, J. Tretmans, and T.A.C. Willemse. A symbolic framework for model-based testing. In *Proceedings of the 1st Combined International Workshops on Formal Approaches to Software Testing and Runtime Verification (FATES/RV '06)*, volume 4262 of *Lecture Notes in Computer Science*, pages 40–54. Springer, 2006.

[14] M. Krichen and S. Tripakis. Black-box conformance testing for real-time systems. In *Proceedings of the 11th International SPIN Workshop (SPIN '04)*, volume 2989 of *Lecture Notes in Computer Science*, pages 109–126. Springer, 2004.

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

[16] G. J. Myers, C. Sandler, T. Badgett, and T. M. Thomas. *The Art of Software Testing, Second Edition*. Wiley, 2004.

[17] L. Nachmanson, M. Veanes, W. Schulte, N. Tillmann, and W. Grieskamp. Optimal strategies for testing nondeterministic systems. *SIGSOFT Software Engineering Notes*, 29(4):55–64, 2004.

[18] M. Newman. Software errors cost U.S. economy 59.5 billion annually, NIST assesses technical needs of industry to improve software-testing. Press Release, `http://www.nist.gov/public_affairs/releases/n02-10.htm`, 2002.

[19] F. Redmill. Exploring risk-based testing and its implications. *Software Testing, Verification and Reliability*, 14(1):3–15, 2004.

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

[21] H. Ural. Formal methods for test sequence generation. *Computer Communications*, 15(5):311–325, 1992.