

Model-Based Test Case Generation for Real-Time Systems

By Anders Hessel

DRAFT: Mon May 7 00:02:14 MEST 2007

This is the titlepage dummy. This page should be substituted for real titlepage. The real titlepage is obtained from the Electronic Publishing Centre after having submitted the template for electronic posting; "spikbladsmallen" to the Electronic Publishing Centre at espik@ub.uu.se. Spikningsmallen (the posting template) can be downloaded from <http://www.ub.uu.se/forauthors/index.php/Main/TemplatesPageEn>.
More information about the publishing routines can be found at <http://www.ub.uu.se/forauthors/index.php/Main/HomePage>

Dissertation at Uppsala University to be publicly examined in Polhemsalen, Ångström laboratory, Lägerhyddsvägen 1, Monday, May 21, 2007 at 13:15 for the Degree of Doctor of Philosophy. The examination will be conducted in English

Abstract

Hessel, A. 2007. Model-Based Test Case Generation for Real-Time Systems. Acta Universitatis Upsaliensis. *Uppsala Dissertations from the Faculty of Science and Technology* 1214. 51 pp. Uppsala. ISBN 91-554-5436-4

Testing is the dominant verification technique used in the software industry today. The use of automatic test case execution increases, but the creation of test cases remains manual and thus error prone and expensive. To automate generation and selection of test cases, model-based testing techniques have been suggested.

In this thesis two central problems in model-based testing are addressed: the problem of how to formally specify coverage criteria, and the problem of how to generate a test suite from a formal timed system model, such that the test suite satisfies a given coverage criterion. We use model checking techniques to explore the state-space of a model until a set of traces is found that together satisfy the coverage criterion. A key observation is that a coverage criterion can be viewed as consisting of a set of items, which we call coverage items. Each coverage item can be treated as a separate reachability problem.

Based on our view of coverage items we define a language, in the form of parameterized observer automata, to formally describe coverage criteria. We show that the language is expressive enough to describe a variety of common coverage criteria described in the literature. Two algorithms for test case generation with observer automata are presented. The first algorithm returns a trace that satisfies all coverage items with a minimum cost. We use this algorithm to generate a test suite with minimal execution time. The second algorithm explores only states that may increase the already found set of coverage items. This algorithm works well together with observer automata.

The developed techniques have been implemented in the tool CoVer. The tool has been used in a case study together with Ericsson where a WAP gateway has been tested. The case study shows that the techniques have industrial strength.

Keywords: Model-Based Testing, Model Checking, Coverage Criteria, Real-Time Systems, Black-Box Testing, Timed Automata, Test Case Generation, Conformance Testing

Anders Hessel, Department of Information Technology, Uppsala University, Lägerhyddsvägen 3, SE-751 05 Uppsala, Sweden

© Anders Hessel 2007

ISSN 1651-6214

ISBN 91-554-5436-4

urn:nbn:se:uu:diva-3344 (<http://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-3344>)

to my family

“Errare humanum est ...”
“Deus ex machina”

Acknowledgements

I would like to thank my supervisor Paul Pettersson for all his efforts during my time as a PhD student. He has been involved in all my research activities, e.g., tool development and paper writing. As Paul's research activities include both testing and real-time systems his expertise and contributions have been invaluable in my studies. Paul has giving me a lot of suggestions for improvements of my texts, both as a co-author of our papers and as supervisor when I have written my licentiate and doctoral theses.

The second person I like to thank is my co-supervisor Bengt Jonsson. He was my (main) supervisor during my first year as a PhD student. I thank him for giving me the opportunity to become a PhD student and for all the help with my master thesis report, which was my first lesson in scientific writing. Bengt is one of the authors of Paper III and has helped me during the writing of my theses.

Wang Yi has been a kind of third supervisor. As the leader of the UPPAAL group he let me into the secrets of the UPPAAL tool and contributed to my understanding of real-time systems, thanks! Besides the supervisors I have had a very fruitful collaboration with the other co-authors of my papers, i.e., Johan Blom (Paper III) and Kim G. Larsen, Brian Nielsen, and Arne Skou (Paper I). I also like to take the opportunity to thank the members of my research group "Testing of Reactive Systems" during my time in the group, i.e., Paul Pettersson, Bengt Johnsson, Johan Blom, Olga Grinstein, Therese Berg, Martin Leucker, Noomene Ben Henda, and Mayank Saksena for constructive discussions. I would like to send collective thanks to all members and former members of the UPPAAL group. Apart from Paul, Wang and Kim I like to mention John Håkansson, Leonid Mokrushin, Alexander David, Johan Bengtsson, and Gerd Behrmann for the help I have got.

The case study in Paper V would not have been possible without our colleges at Ericsson. I owe thanks to Hans Spaak for initiating the contact and to Tomas Aurell for accepting and leading the project at Ericsson. My thanks also go to John Orre, Natalie Jost, Per Vilhelmsson, Payman Tavanaye Rashid, and Joal Dutt for their participation.

I thank the two master students, Fredrik Stenh and Anna Holmgren that have worked with the CO \checkmark ER tool. For those of you that I did not mentioned, you are NOT forgotten.

Last but not least I like to thank my fiancé Anna, just for being you. It is now over and a new epoch of our lives will begin.

List of Papers

This thesis is based on the following papers, which are referred to in the text by their Roman numerals.

- I **Time-optimal Real-Time Test Case Generation using Uppaal.**
Anders Hessel, Kim G. Larsen, Brian Nielsen, Paul Pettersson, and Arne Skou. *In Petrenko A. and Ulrich A., editors, Proc. of the 3rd Int. Workshop on Formal Approaches to Testing of Software 2003 (FATES'03), number 2931 in LNCS, pages 136–151. Springer 2004.*
- II **A Test Case Generation Algorithm for Real-Time Systems.**
Anders Hessel and Paul Pettersson. *In H-D. Ehrich and K-D. Schewe, editors, Proc. of the 9th International Conference on Quality Software 2004 (QSIC'04), pages 268–273, IEEE Computer Society Press, September 2004*
- III **Specifying and Generating Test Cases Using Observer Automata.** Johan Blom, Anders Hessel, Bengt Jonsson, and Paul Pettersson. *In Gabowski J. and Nielsen B., editors, Proc. of the 4th Int. Workshop on Formal Approaches to Testing of Software 2004 (FATES'04), volume 3395 in LNCS, pages 125–139, Springer 2005.*
- IV **A Global Coverage Algorithm for Model-Based Testing.** Anders Hessel and Paul Pettersson. *To appear in Proc. of the 3rd International Workshop on Model-Based Testing 2007 (MBT'07).*
- V **Model-Based Testing of a WAP Gateway: An Industrial Case-Study.** Anders Hessel and Paul Pettersson. *Technical Report 2006-045, Uppsala University 2006.* Extended version of a paper with the same name published in *Brim L., Haverkort B., Leucker M., and van der Pol J., editors Formal Methods: Applications and Technology, FMICS 2006 and PDMC 2006, volume 4346 in LNCS, pages 116–131, Springer 2007.*
- VI **CoVer: A Real-Time Test Case Generation Tool.** Anders Hessel and Paul Pettersson. Submitted manuscript.

Reprints were made with permission from the publishers.

Comments on my Participation

Paper I

I took part in the discussions, did the experiments, and wrote part of the paper.

Paper II

I took part in the discussions, wrote part of the paper, and did the experiments.

Paper III

I took part in the discussions and wrote part of the paper.

Paper IV

I am the principal author of the paper.

Paper V

I am the principal author of the paper. I also modelled the WAP protocol in timed automata.

Paper VI

I am the principal author of the paper. I also implemented the tool CoVer described in the paper.

Other Publications

1. Hessel A., Larsen K., Nielsen B., Pettersson P., and Skou A. (2003) Time-Optimal Test Cases for Real-Time Systems. Invited presentation. In *Proceedings of the 1st International Workshop on Formal Modeling and Analysis of Timed Systems 2003 (FORMATS'03)*, number 2791 in Lecture Notes in Computer Science, pages 234–224. Springer–Verlag 2003.
2. Hessel A., Larsen K., Mikucionis M, Nielsen B., Pettersson P., and Skou A. (2007) Testing Real-time systems using UPPAAL, Book chapter. Accepted for publication.

Summary in Swedish

Värdet av att ett datoriserat system är felfritt är svårt att uppskatta. Felaktigheter i ett system kan utgöra personfara alternativt bli väldigt kostsamma. Testning är den dominerande tekniken för att verifiera att ett system är i enlighet med sin specifikation.

En betydande del av ett programvaruprojekts budget går till testning. Detta är beroende på att planering och utförande av test ofta görs manuellt. Att testa ett system manuellt är ett tidsödande arbete som också är monotont, dyrt och kan göras felaktigt. Därför har många företag automatiserat testningen, kvar är dock problemet att välja testfall. Speciellt problematiskt är det när systemet som ska testas är ett realtidssystem. Då är systemets beteende också beroende av vid vilken tidpunkt som omgivningen ger stimuli till systemet.

Modellbaserad testning är en lovande teknik som stödjer val av testfall. Användaren specificerar här en modell varifrån ett program kan generera testfall automatiskt. En gren inom modellbaserad testing använder en teknik som kallas *modellcheckning* (eng. model checking). I modellchecking går ett program systematiskt igenom en modell för att verifiera egenskaper hos denna.

Ett kvalitativt mått på hur grundligt en testsvit går igenom ett programvarusystem är kodtäckning. Detta kan t.ex. vara en procentsats på antalet programsatser som är exekverade. Andra täckningskriterier kan vara baserade på dataflödesanalys. Vi använder täckningskriterier med avseende på en specifikation i form av en modell, som ledning för val av testfall. På detta sätt får vi en testsvit som testar systemet funktionellt utifrån specifikationen och inte är beroende av hur systemet är implementerat.

De forskningsfrågor som framför allt har upptagit avhandlingsarbetet är hur tekniker som används för modellchecking kan utnyttjas för att generera testsviter och hur täckningskriterier kan specificeras på ett formellt sätt.

Vi börjar med att observera att om vi har ett kriterium som anger att t.ex. varje möjligt tillstånd i en modell ska vara täckt så kan detta delas upp i en uppgift per tillstånd. Vi kallar varje sådan del i kriteriet för en täckningsenhet.

I vår första uppsats beskriver vi hur ett problem för testfallsgenerering kan omdefinieras till ett problem att hitta en väg till ett tillstånd med speciella egenskaper. Vi åstadkommer detta genom att låta ett verktyg för modellchecking söka efter ett tillstånd där alla täckningsenheter är uppfyllda. Detta blir möjligt genom att vi utökar modellen med en boolsk variabel per täckningsenhet och ändrar modellen så att varje sådan variabel har sanningsvärdet sant om och endast dess motsvarande täckningsenhet är uppfylld. När alla vari-

abler har sanningsvärdet sant så har spåret till det tillståndet gått en väg så att motsvarande testfall uppfyller hela täckningskriteriet.

Det modelleringsspråk vi använder oss av heter *tidsautomater* (eng. timed automata). Detta modelleringsspråk kan användas för att modellera realtidssystem. För att inte exekveringen av en testsvit ska ta onödigt lång tid så genererar vi en testsvit som tar minimal tid att exekvera. Detta är ett komplext problem eftersom alla olika testfall kan kombineras för att tillsammans få full täckning.

I avhandlingen presenteras en algoritm som effektiviserar generering av testfall. Vår tidigare algoritm söker ett spår (vari det kan finnas återställningar av systemet) där alla täckningsenheter är uppfyllda. Vi kallar denna typ av generering lokal eftersom ingen hänsyn tas till andra spår. Vår effektiviserade algoritm utnyttjar kunskap om alla spår för att kunna minimera tid och minne, samtidigt som den garanterar att alla täckningsenheter hittas.

När algoritmen hittar en täckningsenhet som aldrig i något spår har hittats tidigare, så sparar algoritmen ett "kandidatspår". När det inte finns något mer att utsöka så kommer det finnas en mängd av kandidatspår som inte överstiger antalet täckningsenheter. Denna mängd reduceras sedan i en andra fas av algoritmen. I och med att alla hittade täckningsenheter finns med i något av kandidatspåren så undanröjs behovet att särskilja tillstånd beroende på tidigare uppfyllda täckningsenheter. Detta leder till en stor reduktion av tillståndsrymden.

Baserat på vår syn på täckning så definierar vi ett språk, i form av parametriserade observatörsautomater, som kan användas för att formellt beskriva täckningskriterier. Vi visar att språket är tillräckligt uttrycksfullt för att beskriva en uppsjö av täckningskriterier som är vanliga i litteraturen. Detta inkluderar kända typer av täckningskriterier såsom logik-, dataflödes- och projektions- (abstraktions) kriterier. Med detta språk definieras också hur tillståndsrymden hos våra specifikationsmodeller behöver utökas för att kunna hitta alla täckningsenheter. Det är de kriterier som användaren definierar som indirekt ger instruktioner till hur algoritmen ska arbeta.

I avhandlingen presenteras också en helt automatiserad testbädd, som inkluderar både urval och exekvering av tester. Vi har använt denna testbädd i en fallstudie av en WAP gateway, som vi gjort i samarbete med Ericsson. Genom att vi gått igenom alla steg från att bygga modell till att få resultat från exekverade tester så har vi visat att tekniken är användbar och fungerar även i en industriell miljö.

Contents

1	Introduction	15
1.1	Real-Time Systems	15
1.2	Testing	16
1.3	Test Case Selection	17
1.4	Model-Based Testing	19
2	Modeling Timed Systems	23
2.1	Untimed State Machines	24
2.2	Timed Automata	25
3	Coverage	27
3.1	Logic Coverage	28
3.2	Data Flow Criteria	30
3.3	Coverage on Projected States	31
4	Research Questions	35
5	Results	37
6	Related Work	43
6.1	Model-Based Testing of Timed Systems	43
6.2	Test Case Generation with Coverage Criteria	45
6.3	Tools for Model-Based Testing	46
7	Conclusion and Future Work	47
7.1	Conclusion	47
7.2	Future Work	48
	References	51

1. Introduction

The fact that unreliable computer systems can cause severe problems in our society is indisputable. Apart from the personal and material damage an incorrect system can cause to its user or owner, it can also be costly for the manufacturer. For these reasons manufacturers strive to make their systems as error-free as possible.

For a system to function correctly, there are two things that are important: *validation*, to ensure that the right system is built, and *verification*, to ensure that the system is built right. In this thesis we will consider the verification problem.

Testing is the dominating verification method for increasing confidence in a computer system. It is the process of exercising a system in a controlled environment and examine if its behavior complies with the requirements of a system. There are other quality improvement techniques used by software engineers as part of the verification process. Other techniques include code walk throughs, code inspections, and code reviews.

The purpose of testing is to reveal *faults* in the system. Testing can only show the presence of faults, not their absence. There are two main challenges in testing, to *select* and to *execute test cases*.

We will consider testing of the logical and temporal correctness of a system, i.e., functional testing. There are many other types of testing. Among them *stress testing*, which is often used interchangeably with both *load testing*, and *performance testing*, i.e. testing when the system is heavily utilized. A *duration test* is a test of the ability of a system to run over a longer period of time, and *robustness testing*, sometimes called *negative testing*, conducted by sending invalid input data, are also outside the scope of the thesis.

1.1 Real-Time Systems

A *real-time system* is a system where the behavior of the system depends not only on the input but also on the timing of the input. Such system can also have requirements on the timing of its outputs. In order to test a real-time system, we have to take into account not only *what* inputs to supply to the system, but also *when* to supply them. For correct behavior of a real-time system, a response should not only provide correct values, but the values should also be provided at the right time-points.

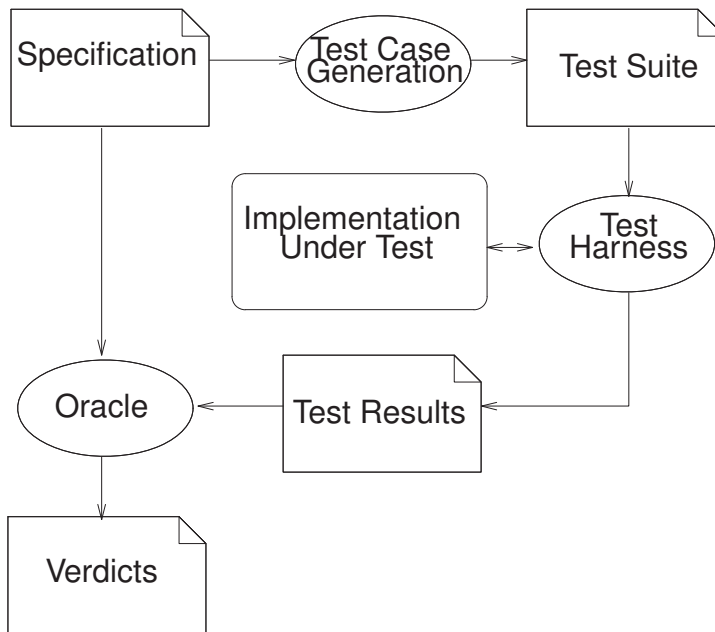


Figure 1.1: Black box testing

We illustrate a few different timing requirements. (i) A machine that is filling up soda bottles must stop after a certain amount of time in order not to overflow. Hence the specification of an embedded controller of the machine shall require an output from the controller to stop filling after that time.

(ii) A computer that distinguishes between single- and double-clicks from an input device must measure the time between two consecutive clicks. First after waiting the maximum time bound for a double-click, the computer can determine a first click as a single-click. If a second click arrives earlier, then the two clicks should be interpreted as a double-click.

(iii) A car control system might have to react to a brake signal within a given time bound. During the reaction time, the engine ignition must still be on time with the requested precision.

1.2 Testing

Testing of system behavior can be categorized into white box and black box testing. In *white box testing*, also called *structural testing*, or *glass box testing*, tests are derived from knowledge about the structure of the software and from implementation details. In *black box testing*, also called *functional testing*, test data are derived from the specified functional requirements without consider-

ing the internal program structure [ABC82]. The implementation and the test cases can be developed in parallel, by two separate teams.

In Figure 1.1, a common setting for black box testing is shown. From a *specification* of the system, *test cases* are derived. A test case includes input values that stimulate the system to test some chosen functionality. This could be parameters to start the system or sequences of input data. For real-time systems also the timing of the input data should be supplied. The *test case generation* results in a collection of test cases called a *test suite*. The generation may be done manually or automatically.

After the test suite is produced, a *test harness* executes the test suite against the implementation under test. This produces a *test result*, which is compared to the expected result, prescribed by the specification, by a *test oracle*. The test oracle delivers a *verdict* for each test case in the test suite. Ideally, the verdict of a test should be *pass* or *fail*. If all generated tests pass, then this shows conformance between the test and the specification.

A failed test is a *system failure*, i.e., the system does not deliver the expected result (erroneous or with incorrect timing). If the test is carried out under the specified circumstances, then the failure shows that the system has an *error*, i.e., a design flaw. When a test fails we identify the *fault* (the defect) causing the failure.

If a test has failed, the system (as a whole), does not conform to the test. The test itself might not conform to the specification, and in that case the test case should be changed and not the system. Further, the specification may not express the intention of the system. In this case the the specification may be changed and the test cases rewritten.

Often the expected test result can be incorporated into the test cases so that the test harness can make the verdict itself; in this case the oracle is a part of the test harness. This is especially good if the test cases consist of long sequences, because the test harness can stop further interaction and execute the next test case if it discovers an error.

1.3 Test Case Selection

For a program (or a function) that takes a finite set of input parameters, each of which has a specified input domain, testing all combinations of parameter values is referred to as *exhaustive testing*. The number of parameter combinations can be very large, which makes exhaustive testing not applicable in most practical cases.

One way to reduce the number of test cases in a test suite, and still test all functionality in a specification, is by using *partition testing*. In partition testing the value domains are divided into *equivalence classes*, and the tests are selected so that at least one value from each equivalence class is tested. Any values within the specified value domains can be chosen arbitrary. For

robustness testing, also values that exceed the minimum or maximum bound of the domains can be chosen. *Boundary values* or *extreme values* are the values that lie close to the border between valid and invalid data. It is typically very interesting to select boundary values for tests.

If a system can receive arbitrarily long input sequences, and has an internal state that is updated after each input then exhaustive testing is not possible. Two examples of such systems are (i) a compiler that reads a source code file as input (stream) and (ii) an elevator controller that repeatedly reacting on input events. In addition to input sequences being arbitrarily long, the timing of the inputs matter for a real-time system.

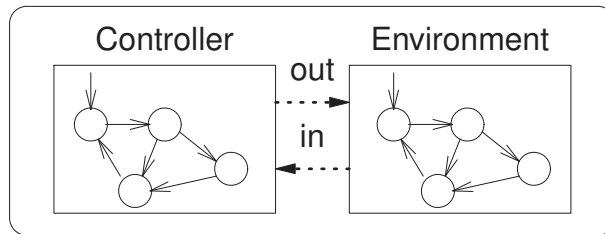
If the internal structure of a system is known, as in white box testing, test cases can be generated with knowledge of the actual code that is exercised during test execution. It is possible to make one test suite tailored to test one specific part of the code. The code can be instrumented to report which lines are exercised, e.g., by using the *gcov* tool [vHW03] or EMMA [Rou06]. A test suite can be said to cover partially or fully the code with respect to some measure of *coverage*, e.g., use of every statement or every branch in the code. Such *code coverage* is typically used to measure the thoroughness of a given test suite. It assists engineers to improve their test suites by pointing out the parts not exercised. In Figure 1.1, the test suite would be affected by the specific implementation if white box testing is used.

A *test purpose* is a specific objective (or property) that the tester would like to test, and can be seen as a specification of a test case. Test purposes can be used to select test cases. As an example of a test purpose, we consider “test of a state change from state A to state B” in a specification. For this purpose a test case should be generated that covers the specific state change. If we make a test purpose for all specified state changes, and generate test cases for them, then we have a test suite that covers all specified state changes in the specification. As a test case can cause several state changes, and thus fulfill several test purposes, a test suite might have fewer test cases than the number of test purposes it fulfils.

Non-deterministic specifications can be used if the cause of some decision is unknown or the details that determine the decision are abstracted away. Because of the non-determinism, we will not always have one possible response from an implementation, but several. We can use *adaptive test cases*, which requires that the test harness has a decision tree for each test case.

If a test purpose is to exercise a particular state change in the model, and we make a test case for this state change, then we cannot be sure that we will succeed (even with a correct system), if the specification allows non-determinism. A decision tree can have arbitrary long branches without any guarantee of capturing the desired behavior. When we reach a leaf of a decision tree, we still might not have been able to exercise the desired behavior. We cannot give the verdict fail, which would indicate that the system is non-conformant with the test specification. Still, the test purpose is not fulfilled, and thus the verdict

Model of System and Environment



... is an abstraction of ...
System during Test Case Execution

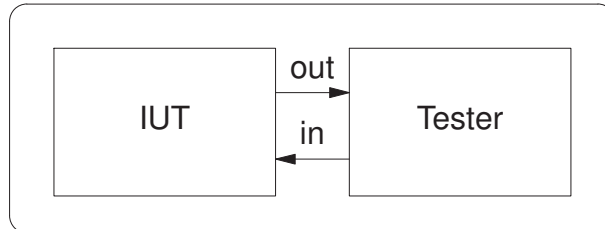


Figure 1.2: Model-based testing

pass would be misleading. In this case we give the test the verdict *inconclusive*. We can run this test again and we may get another result.

1.4 Model-Based Testing

Model-based testing is a black box testing technique where test cases are derived from a model that specifies the expected behavior of a system. Formal models with precise semantics are of extra importance, because they are suitable for automatic test case generation. In this thesis we regard a model as a state-based formalism. This type of model-based testing is also called *state-based testing*.

A common distinction in model-based testing is to separate the specified system that is called *controller* and the environment of the systems that is called *environment*. The relation between the models during exploration and real world testing is shown in Figure 1.2. The upper part shows the model partitioned as described above, and the lower part shows the implementation under test (IUT) and the tester, e.g., the test harness or a human user. The communication channels between the controller and environment in the modeling world have a corresponding communication medium in the real world. It is possible to create a fully open environment for the controller. This is achieved

if the environment can send (and receive) any stimuli at any time. Such unconstrained environment have the ability to be stateless.

Regression tests are tests that are executed after an update of a system. It can be very costly to update the test cases manually after a change in the specification. If model-based testing is used new test suites can be generated after updating the model.

The most ambitious efforts to show conformance between a system and a model is to check for every state and stimuli that the system makes the proper transition. For each transition a test case drives the system to the source state of the transition, makes the system perform the transition, and finally check that the system is in the target state [Cho78, LY96, LPU02]. For more complex systems this method experience the state explosion problem. To avoid state space explosion other more selective methods, concentrating on test purposes or coverage criteria, have been studied.

In structural testing the code coverage can be used as a measure of thoroughness for a test suite. In the same way coverage on the model can be used as a measure of thoroughness for a test suite in model-based testing [BGH⁺99, BAKD01]. Such a measure can be evaluated without execution of the tests against an implementation. Test purposes can be used also for models [JJ05]. A system is constructed by parallel composition of an environment model expressing the purpose and the controller model. If the environment can reach a special state the trace will be a test case that fulfills the purpose.

State-space exploration engines needed in model-based testing is a built-in feature of a model checker. A *model checker* can formally verify temporal properties of a system model. Reachability properties is one type of properties that a model checker can verify. A *reachability property* specifies that a state with a certain property should be reachable, e.g., “There exists a reachable state s so that P holds for s ”. A state is defined as *reachable* in a model, if it can be reached from the initial state by zero or more transitions.

One way for a model checker to check reachability is to explore the reachable states from the initial state, either until it finds a state where the property holds or until there are no more states to explore. A path to such a state is called a *witness trace*. It is a trace from the initial state to a state where the property holds.

It is possible to transform the problem of test case generation into a reachability problem. The main idea is to use a reachability property to determine if a test purpose can be fulfilled. In [HLSU02, HCL⁺03], Hong et al use a model checker to generate test cases for data-flow criteria, e.g., definition-use pairs. In the case of definition-use pairs each pair is described as a property and (if the purpose is fulfilled) the model checker returns a witness trace for that particular pair. A test suite for the total criterion is thus the collection of the traces form all pairs.

If a test suite satisfy a coverage criterion, it guarantees that the test suite exercise the system with a certain thoroughness. This can be considered as

a quality measure, but is not the only way to measure the quality. Much efforts have been spent to produce test cases that distinguish a *mutant* from the original specification. A mutant is a program that is similar to the correct specification but is slightly changed in some way, e.g., a logic AND instead of a logic OR in a decision. Mutant analysis have been practiced by e.g., Ammann et al [ABM98].

2. Modeling Timed Systems

In model-based testing a model is used as the specification. A model is an abstraction of a desired system behavior, which can consist of the combined behavior of applications, OS, hardware etc. Benefits of modeling includes understanding of the specification in an early stage, and exposition of ambiguities in the specification and the design. For some types of models, model checking tools (such as SPIN [Hol97] or UPPAAL [LPY97]) can be used to formally verify properties of the model, in order to find errors in a model before implementing it. Errors found late in a project are known to be more expensive than errors found early. Thus, it is important to make a correct specification.

The abstraction process results in a model. Different types of abstractions often have to be made to construct a model. For example, an integer variable that is used only by a model in a decision that evaluates whether the value is odd or even, can be reduced to two abstract values “integer-odd” and “integer-even”. An IP-number used when communicating with a system is typically something that can be abstracted away completely.

The right level of abstraction is crucial for any model-based technique to be successful. If a model is too abstract, the targeted functionality cannot be tested, because details important to distinguish different cases are missed. If the model is too concrete, the construction of the model is as error prone as a full implementation. In state-based testing the control states are in focus and data that does not affect the control behavior can be abstracted away e.g., the payload in a network protocol.

An abstract test case generated from a model is on the same abstraction level as the model it is derived from. If we have an abstract test case with the abstract values “integer-odd” and “integer-even” for a variable, then we can replace “integer-odd” and “integer-even” with, e.g., 0 and 1. Only when all abstract values have been replaced with concrete values, the test case can be executable. Other data that have been completely abstracted away must of course also be added, e.g., IP-numbers.

For a given system, it is possible to produce many models of different aspects. If there are different functionalities that are orthogonal to each other, it is often easier to validate each functionality in a separate model than to validate a combined model. To have a valid model is the base requirement for any model-based technique.

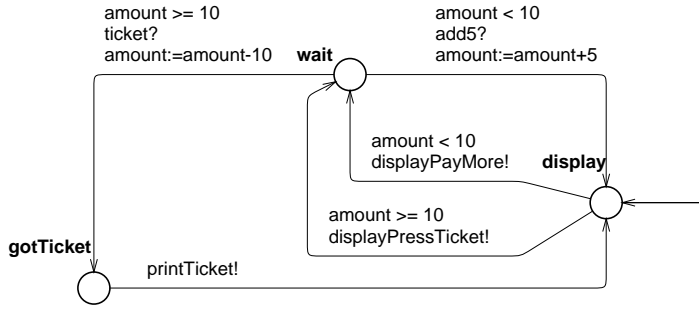


Figure 2.1: EFSM model of a parking ticket machine.

2.1 Untimed State Machines

In this section we will introduce state machines without time. A *finite state machine* (FSM) is an abstract machine with a finite set of *locations* L , a finite set of *edges* E , and a finite set of *actions* Act . One of the locations $l_0 \in L$ is the initial location. An FSM uses actions to interact with its environment. An edge is a triple $(l, \alpha, l') \in E$ that has a source location $l \in L$ and a destination location $l' \in L$ and is labeled with an action $\alpha \in Act$.

In our case the actions can be partitioned in input actions, output actions, and internal actions. We will use the convention that an input action is suffixed with “?”, and an output action is suffixed with “!”. An internal action has no suffix.

An *extended finite state machine* (EFSM) consists of locations L , edges E , actions Act , and *variables* V . The location $l_0 \in L$ is the initial location. Each variable x has a value domain. An edge is a quintuple $(l, g, \alpha, u, l') \in E$ that has a source location $l \in L$ and a destination location $l' \in L$ and is labelled with a *guard* g , an action α , and an *update* u . The guard g is a predicate over V , and the update u is an assignment where each variable v is assigned a value from an expression over V . If there is no assignment the variable values are unchanged.

A state of an EFSM is a tuple $\langle l, \sigma \rangle$ where $l \in L$ and σ is a mapping from V to values. The initial state is $\langle l_0, \sigma_0 \rangle$ where σ_0 is the initial mapping. A transition between two states, i.e., from $\langle l, \sigma \rangle$ to $\langle l', \sigma' \rangle$ is possible if there is an edge $(l, g, \alpha, u, l') \in E$ where the g is satisfied for the valuation σ , σ' is the result of updating σ according to u , and α is an action that require communication with the environment, if the action is not internal.

If we assume that every variable has a finite domain in an EFSM, then the EFSM can be viewed as a compact notation of an FSM. It is possible to unfold the EFSM such that each EFSM state is an FSM location and each EFSM transition is an FSM edge.

In Figure 2.1 an EFSM modeling a parking ticket machine is shown. The initial location is named *display*. There is one variable *amount* that is initially 0. As a first transition the EFSM can only output the action *displayPayMore* that models displaying the message “Pay more”. In the *wait* location, the user can add 5 credit coins that increment the value of *amount*. If the user has paid enough for a ticket, the EFSM outputs the action *displayPressTicket*, else it outputs the action *displayPayMore*. After an output *displayPressTicket* from the EFSM the user can input *ticket* and get an output *printTicket* in return. Note that the *wait* location is used in three different states $\langle \text{wait}, 0 \rangle$, $\langle \text{wait}, 5 \rangle$, and $\langle \text{wait}, 10 \rangle$. In the first two the user is allowed to add more credits, and in the last the user is allowed to request a ticket.

2.2 Timed Automata

We use *timed automata* [AD94] to model timed systems. Let X be a set of non-negative real-valued variables called *clocks*. Let $\mathcal{G}(X)$ be a set of guards on clocks generated by the grammar

$$g ::= x \bowtie c \mid x - y \bowtie c \mid g_1 \wedge g_2$$

where $x, y \in X$, $c \in \mathbb{N}$, and $\bowtie \in \{<, \leq, =, \geq, >\}$. A timed automaton consists of locations L , edges E , actions Act , and clocks X . One of the locations $l_0 \in L$ is the initial location. An edge $(l, g, \alpha, r, l') \in E$ has a source location $l \in L$ and a destination location $l' \in L$ and is labelled with a guard $g \in \mathcal{G}(X)$, an action $\alpha \in Act$, and set of clocks to reset $r \subseteq X$ called *reset*.

A state of a timed automaton is a tuple $\langle l, \sigma \rangle$ where $l \in L$ and $\sigma \in \mathbb{R}_{\geq 0}^X$ is a mapping from X to non-negative real-time values. The initial state is $\langle l_0, \sigma_0 \rangle$ where σ_0 is the initial mapping where every clock is mapped to 0. There are two kinds of transitions, *discrete* transitions and *delay* transitions. A discrete transition between two states written $\langle l, \sigma \rangle \xrightarrow{\alpha} \langle l', \sigma' \rangle$ is possible if there is an edge $(l, g, \alpha, r, l') \in E$ where the guard g is satisfied for the valuation σ , where r is an update so that $\sigma' = \sigma[x/0]$ for all $x \in r$, and α is an action. In a delay transition between two states written $\langle l, \sigma \rangle \xrightarrow{d} \langle l, \sigma + d \rangle$, where $d \in \mathbb{R}_{>0}$, and $\sigma + d$ denotes the result of incrementing all clock values in σ with d . Locations can have invariants, that set an upper bound on a clock value. The bound constrains the delay so that the automaton is not allowed to stay in the location forever. Timed automata use *dense time*, which means infinite precision of clocks.

In Figure 2.2 a timed automaton modeling an explosive pen is shown. A double-click, i.e., an input *click* twice within the time bound C , will arm the pen to explode after B time units. Here B and C are positive integers. An armed pen can be unarmed by another double-click if the double-click is supplied before the explosion. The explosion is modeled in the automaton by the

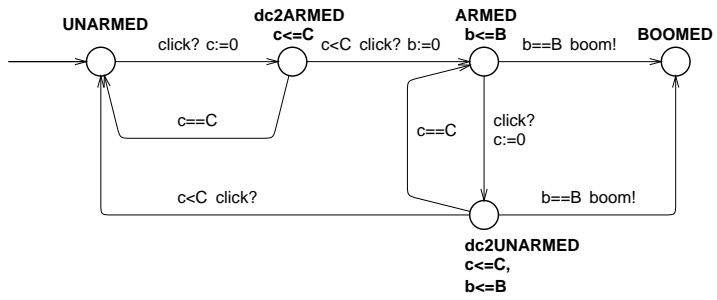


Figure 2.2: Timed automaton describing the function of the explosive pen in the movie Golden Eye

output *boom*. If the pen works correctly it could be used for its purpose safely. Needless to say, incorrect use could be devastating.

3. Coverage

Time and money are commonly used criteria to determine whether to end the testing of a product or not. Unfortunately these criteria do not set any quality standard on the product. When should then a system be considered to be tested thoroughly enough? Without any objective measure this is a hard question. If parts of the system are not exercised at all, then the system is probably not tested thoroughly enough.

A test suite can be measured with respect to the amount of code it exercises, e.g., by measuring the number of statements executed in the system under test. The number of statements exercised can be compared with the total number of statements, and the percentage can be calculated. A criterion for enough thoroughness can be to exercise a certain percentage of the number of statements. Here we use statements as our measure, but we could also use the exercised branches between statements. If we use the statement measure, a test suite is said to *cover* a statement, if the statement is exercised (at least once) during an execution of the test suite. Coverage with respect to the statement measure is called *statement coverage* [Mye79], and coverage with respect to the branch measure is called *branch coverage* [Mye79]. Statement coverage and branch coverage are examples of *coverage criteria* that can be used to measure coverage provided by a test suite.

There are many other coverage criteria. We will give an overview of several other coverage criteria later in this section. For this purpose we will use the terminology from Paper III of this thesis, to explain the coverage criteria. We assume that all coverage criterion consists of a set of measurable items. We will use *coverage item* as a generic term for a measured item. For statement coverage, exercising a statement will fulfill a coverage item. There will be one coverage item for each statement. Thus, to achieve full (statement) coverage all statements must be exercised.

It may not be possible to list all (feasible) coverage items without sophisticated analysis of the code. Even then, a coverage criterion must describe how to identify a coverage item and how to distinguish coverage items from each other.

For model-based testing, analysis of coverage with respect to a criterion, can be used to guide test suite generation. When a black box test suite is executed, the actual code coverage can be measured. Based on the result, additional test cases can be added. White box measures can thus complement test suites generated with black box techniques.

In an EFSM (or other models as FSM or TA) to visit all locations is called *location coverage*, and to traverse all edges is called *edge coverage*. In the remainder of this section, we first describe some classic logic-coverage criteria in their original context of code. We then describe data flow and projected state coverage in an EFSM context.

3.1 Logic Coverage

White-box testing is concerned with the degree of thoroughness to which test cases exercise the logic (or source code) of the program. We distinguish between *decisions* that decide the continuation of the program control and *statements* that are non-branching. In an if-statement (in a C-like syntax)

$$\text{if } (x = 2 \wedge y \geq 6) s_1; \text{else } s_2;$$

the decision is “ $x = 2 \wedge y \geq 6$ ”. It decides which of the branches of statements, s_1 or s_2 , the program control should follow. Subexpressions that do not contain \wedge , \vee , or \neg are called *conditions*, e.g., $x = 2$ and $y \geq 6$. The execution order or branching inside a decision is not considered.

We have already described the statement coverage criterion, which requires a test suite to execute each statement in the system under test. If 100% coverage cannot be achieved, then there must be some dead code in the implementation. Statement coverage is similar to *line coverage* or *basic block coverage*. In basic block coverage a sequence of non-branching statements is the measured unit, but because basic blocks are non-branching, basic block and statement coverage are equivalent metrics, given that full blocks are always executed. In statement coverage each statement corresponds to a coverage item. The statement identifies the coverage item.

We have also described *decision coverage* (DC), but under the alternative name *branch coverage*, which stipulates that each possible branch must be traversed, e.g., both the true and the false branch must be traversed for an if-statement. In a switch-statement all cases must be traversed. This means that the decision expression is considered as one unit without considering its conditions.

For a decision, e.g., $c_1 \vee c_2$ in an if-statement, where c_1 and c_2 are conditions, DC can be achieved by two test cases where the conditions evaluate to $\{c_1 = \text{true}, c_2 = \text{false}\}$ and $\{c_1 = \text{false}, c_2 = \text{false}\}$, i.e., the truth value of condition c_2 is not changed. In DC each outgoing branch from each decision corresponds to a coverage item. The branch identifies the coverage item.

The *condition coverage* criterion (CC) [Mye79] is sensitive to each condition as it require all possible outcomes of each condition in a decision. In general CC is a stronger criterion than DC, but this is not always true. For a decision, e.g., $c_1 \vee c_2$ in an if-statement, CC can be covered by test cases where the conditions evaluate to $\{c_1 = \text{true}, c_2 = \text{false}\}$ and $\{c_1 = \text{false}, c_2 = \text{true}\}$.

As both evaluations of $c_1 \vee c_2$ become true, DC is not fulfilled and the false branch will never be traversed. In CC a coverage item is identified by a tuple $\langle c_i, b \rangle$, where c_i is a condition in the code, and b defines the truth value of c_i , i.e., there are two coverage items for every condition.

The *multiple condition coverage* (MCC) [Mye79] requires that all possible combinations of outcomes of the conditions in each decision must be exercised. Let d_i be an index that identifies the i^{th} decision, n_i be the number of conditions in d_i , and K_i be a tuple of truth values of the possible outcomes of the conditions of d_i . In MCC a coverage item is identified by $\langle d_i, k_j \rangle$, where $k_j \in K_i$ is an outcome of the conditions of the decision d_i .

A relaxation of the MCC criterion is the modified condition/decision coverage (MC/DC) criterion created by Boeing [RCT92, CM94]. It requires (for each decision) every condition to modify the outcome of the decision without changing the truth values of the other conditions in the decision. As the modifying condition must change the outcome of the decision without changing the other truth values MC/DC *subsumes* both DC and CC.

In MC/DC a coverage item is identified by $\langle d_i, c_j \rangle$, where c_j is a condition in a decision d_i , so that c_j has made the decision d_i both true and false without changing the other conditions in d_i . We will use the intermediate information $\langle c_j, k_j, \text{true} \rangle$ and $\langle c_j, k_j, \text{false} \rangle$, where k_j is a tuple of truth values for the other conditions than c_j in d_i . Because of the requirement that the other conditions in the decision d_i are not allowed to change their truth values $\langle c_j, k_j, \text{true} \rangle$ cannot exist without $\langle c_j, k_j, \text{false} \rangle$. This might look a lot like CC, but we have different conditions for the coverage items. Again for a decision, e.g., $c_1 \vee c_2$ in an if-statement, CC can be covered by test cases where the conditions evaluate to $\{c_1 = \text{true}, c_2 = \text{false}\}$ and $\{c_1 = \text{false}, c_2 = \text{true}\}$. All the possible CC coverage items are covered, i.e., $\langle c_1, \text{true} \rangle$, $\langle c_1, \text{false} \rangle$, $\langle c_2, \text{true} \rangle$, and $\langle c_2, \text{false} \rangle$. If MC/DC is applied for the same test suite no cover items are covered. If we add a case where $\{c_1 = \text{false}, c_2 = \text{false}\}$, then MC/DC will be covered, i.e., $\{c_1 = \text{true}, c_2 = \text{false}\}$, $\{c_1 = \text{false}, c_2 = \text{false}\}$ will give $\langle c_1 \rangle$, because we have $\langle c_1, k_1, \text{true} \rangle$ and $\langle c_1, k_1, \text{false} \rangle$ where $k_1 = (\text{false})$, only c_1 change. The second coverage item $\langle c_2 \rangle$ (only c_2 change) is covered by $\{c_1 = \text{false}, c_2 = \text{true}\}$, $\{c_1 = \text{false}, c_2 = \text{false}\}$. Note that the decision must be visited twice to fulfill one coverage item. It is possible that this requires two test cases.

Switch coverage [Cho78], is a classic coverage criteria. The possible path of control flows can not only split up in decisions they can also join, e.g., after an if statement. A “switch” is a combination of the entrance and the exit branch of a basic block. If a basic block has two incoming and two outgoing branches there are four possible “switches”. Put it another way, a coverage item of the switch coverage criterion is a pair $\langle b_{in}, b_{out} \rangle$, where b_{in} and b_{out} are branches between basic blocks. The coverage item $\langle b_{in}, b_{out} \rangle$ is fulfilled if the branch b_{in} is the entrance and b_{out} is an exit of a basic block. This can be extended to n -tuples where n consecutive branches are exercised to fulfill the coverage item.

3.2 Data Flow Criteria

Data flow testing criteria [CPRZ89] are among the most common criteria. We first make some definitions. A variable is *defined* when it is assigned a new value, e.g., c is defined in the statement $c := k + 1$. A variable is *used* when it is part of a computation or a predicate. An example of a *computation-use* (c-use) is k in the statement $c := k + 1$, and an example of a *predicate-use* (p-use) is x in a guard $x = 1$. A variable has a p-use in all types of decisions. When we refer to the locations of the definitions or of the uses we will use EFSM terminology, i.e., both assignments and guards are located at edges. Without loss of generality we restrict the number of assignments on an edge to one in the further presentation, thus an assignment can be referenced by an edge. A *path* is a sequence of edges traversed by an automaton consecutively.

Assume a variable x is defined at edge e_d . Consider a path $e_d, e_1, \dots, e_n, e_u$, where the variable x is not redefined in the sub-path e_1, \dots, e_n , then the definition at e_d *reaches* the edge e_u . In this case the sub-path e_1, \dots, e_n is a *definition-clear path* with respect to x .

Reach coverage [Her76, LK83], also referred to as definition-use pair (du-pair) coverage is a commonly used criterion. It requires that a test suite includes all paths from a definition of a variable x to all the reachable edges where x is used. A coverage item $\langle x, e_d, e_u \rangle$ where x is a variable, e_d is an edge where variable x is defined, and e_u is an edge with a use. If a definition of a variable x at edge e_d reaches a use of x at edge e_u with a definition-clear path with respect to x , then the coverage item $\langle x, e_d, e_u \rangle$ is fulfilled. Notice that two different variables x and y can both be defined in e_d and used in e_u still $\langle x, e_d, e_u \rangle$ and $\langle y, e_d, e_u \rangle$ would not always be covered for the same paths. This is because a definition-clear path from e_d to e_u with respect to x might not be a definition-clear path with respect to y and vice versa.

Another criterion is *context coverage* [LK83] or rather *definition context coverage*. A *context* of a variable definition is the edges where the variables used for the definition is defined, e.g., for an assignment $x := y + z$ the context is $\langle e_y, e_z \rangle$ if y was defined at e_y , and z at e_z , when x is defined. The criterion requires that a test suite includes all paths so that for every definition of a variable x , every different context of the definition is represented. A coverage item for context coverage is a pair $\langle e_i, k_i \rangle$ where e_i is an edge with a use, and k_i is a tuple of the edges where the variables used in e_i is defined. The size of k_i depends on the used variables in e_i . For used variables v_1, \dots, v_n in e_i , k_i consists of e_1, \dots, e_n so that v_1 is defined in e_1 etc.

A similar criterion is *ordered context coverage* [LK83]. An *ordered context* of a variable definition is a context where the edges in the context is listed in the order of their definition, e.g., for a statement $x := y + z$ the ordered context is $\langle e_1, e_2 \rangle$ where y was defined at e_2 , and z at e_1 , and the definition of y is the more recent of the two. A coverage item for ordered context coverage is

similar to one in context coverage with the difference that the positions in the tuple k_i are sorted in the order by the occurrence in the path to e_i .

The *all-paths* [RW85] criterion is fulfilled if all possible paths are included in a test suite. This is not feasible for general EFSMs that can have arbitrary long paths or for code that has infinite loops. Here, we consider only systems with a finite number of paths with a specific entry and exit point. The coverage criterion is fulfilled if all paths from the entry to the exit point are included in the test suite. A coverage item for the all-paths criterion consists of a single parameter p that is a full path from an entry point to and exit point.

The *all-defs* [RW85] criterion is one of the easiest data flow criterion to fulfill. It is enough to find one use for each definition. If we have a path that contains a definition of a variable that reaches a use, then another path that contains another use of the same definition does not cover any new coverage item. A coverage item for the all-defs criterion $\langle x, e_d \rangle$ requires that a definition at edge e_d of a variable x has a definition-clear sub-path with respect to x to an edge where x is used.

The *all-uses* [RW85] criterion is another name for reach coverage. The *all-p-uses* [RW85] criterion is similar to all-uses, but the use at e_u must be a p-use. The *all-c-uses* [RW85] criterion is similar to all-uses, but the use at e_u must be a c-use.

The *all-du-path* [RW85] criterion requires that all (definition-clear) paths between a definition and a use with respect to the variable are included in the test suite. A coverage item $\langle x, p \rangle$ is covered if there exists a definition-clear path p with respect to a variable x , where p starts with an edge where x is defined, and ends with an edge where the variable is used. The all-du-paths criterion is stronger than all other definition-use criteria, because every du-path must be covered.

In [Nta88], the *Ntafos' required k-tuples* criteria are described. A *2-tuple* is simply a du-pair. We denote a du-pair $\langle x_1, e_1, e_2 \rangle$, where the variable x is defined at edge e_1 and used at edge e_2 . If at e_2 , the variable x_1 affects the definition of x_2 in another du-pair $\langle x_2, e_2, e_3 \rangle$, then the two du-pairs are here said to be *coupled*. Two coupled du-pairs forms a 3-tuple. Note that for a 3-tuple there are three edges involved. The criterion, for k-tuples, requires that a test suite includes paths so that every k-tuple is included.

A coverage item $\langle e_1, x_1, \dots, x_{k-1}, e_k \rangle$ for $k > 2$, where a variable x_1 is defined at edge e_1 , a variable x_{k-1} is used at edge e_k , the edge e_i where $2 \geq i \geq k - 1$ is an edge where a variable x_{i-1} affects a another variable x_i , and there is a definition-clear path from e_j to e_{j+1} with respect to x_j where $1 \geq j \geq k - 1$. Thus the definition at edge e_0 affects the use at edge e_k .

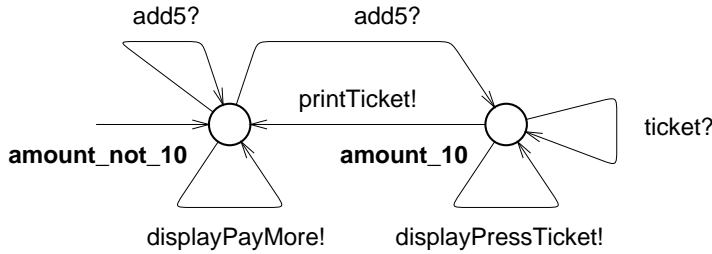


Figure 3.1: A projected state machine graph. The ticket machine in Figure 2.1 has been projected on the amount variable.

3.3 Coverage on Projected States

We have earlier described location coverage, i.e., to visit all locations in the model. In an EFSM, states are defined by locations together with valuations of variables. If we simply take the location of a state, then we have made a projection from a state to a location.

Let M be an EFSM. We can describe the operational semantics of M as a labeled directed graph $G = (V, E)$, where vertices V corresponds to the states of M and the labeled arcs E corresponds to the transitions of M where the label is the action used. Let $\lambda : V \rightarrow V'$ be the projection function that maps states in V to states in V' . As an example $\lambda_{loc} : V \rightarrow L$, where L is the set of locations in M , maps a state to the location of the state, e.g., $\lambda_{loc}(\langle l, \sigma \rangle) = \langle l \rangle$. If $\lambda(v) = \lambda(v')$, where $v, v' \in V$ then v and v' are in the same equivalence class.

Let ρ be an equivalence relation and $[v]$ denote an equivalence class, where $v \in V$. A *projected state machine graph* [FHNS02] under a equivalence relation ρ is defined to be the labeled directed graph $G' = (V', E')$, where V' is the set of equivalence classes under ρ , and E' is the set of labeled arcs, where an arc labeled a from $[v]$ to $[v']$ in E' exists if $\exists s, s'. \lambda(s) = [v] \wedge \lambda(s') = [v'] \wedge \langle s \xrightarrow{a} s' \rangle \in E$.

In Figure 3.1 a projected state machine graph is shown. We have projected the parking ticket machine from Figure 2.1 on the variable *amount*. The equivalence classes are $amount = 10$ and $amount \neq 10$. Notice that the projection is non-deterministic, the *add5* action is associated with two arcs without restrictions. As there is no other information in the projected states than the equivalence class the target projected state is non-deterministic.

If we make a test suite that cover every arc in the projected state machine graph, we note that there are two arcs labeled *add5?* to be covered, whereas there is only one edge with *add5?* in the original model. A test suite must cover both the case where a five credit coin is added and the total amount will not sum up to ten credits and the case when the total credit sums up to ten.

If we project the parking ticket machine from Figure 2.1 on equivalence classes that are separated by both the location and the *amount* variable, we would get the full state-space of the ticket machine. If there were more variables used in the ticket machine the projection might have been useful to concentrate on the behavior around the *amount* variable.

4. Research Questions

As described in Section 1.4 model-based test case generation is a technique to assist a test engineer to systematically select test cases from a model. To select test cases so that a coverage criterion is satisfied, the model needs to be systematically explored. Such exploration can be done with a model checker. In reachability analysis a model checker explores all possible states of a system in order to check whether a state with a given property can be reached from the initial state.

Model checking techniques for untimed systems have been used to produce test cases from a single test purpose. Is it possible for a model checker for timed systems to produce a test suite that satisfies a coverage criterion? We formulate our first research question:

How can model-checking techniques for timed systems be used for test-suite generation?

Coverage criteria have often been described in natural language. For the purpose of test case generation, model checkers have been used for single test purposes (coverage items) characterized with temporal logical formulas. Ideally, a formal language that is intuitive, flexible, and independent from the model language should be used for coverage criteria specification. If so, an algorithm that takes a model of a system under test and a coverage criterion should be able to generate a test suite fulfilling the coverage criterion. We formulate the research question:

Is it possible for an algorithm to accept a specification of a coverage criterion and a model of a system, and generate a test suite accordingly?

In our research group we have access to the code of a model-checking tool. We suspect that the reachability algorithm found in a model checker is not the most efficient algorithm for test case generation. If we utilize the fact that our only interest is to generate test suites, can we then improve the algorithm, so that it uses less time and memory? We express this as a research question.

How can model-checking techniques for timed systems be specialized for test-suite generation to use less time and memory?

Our research project was at first sponsored by the VINNOVA competence center ASTEC (Advanced Software Technology) and our research efforts were in cooperation with industrial partners. This gave us the opportunity to focus on techniques that are implementable and of practical use in an industrial environment. We formulate the research question:

Is it possible to apply model-based specification and generation techniques to test an industrial-sized timed system?

5. Results

We have tried to answer the questions from the previous section in six research papers. The first research question is mainly addressed in Paper I:

How can model-checking techniques for timed systems be used for test-suite generation?

In the paper, we show how the model-checker tool UPPAAL can be used to generate test cases for three common coverage criteria. We also define a class of automata needed to get reproducible test traces. In Paper III we address the second research question:

Is it possible for an algorithm to accept a specification of a coverage criterion and a model of a system, and generate a test suite accordingly?

In the paper we present a test case generation algorithm that from an EFSM model and a coverage criterion generates a test suite that satisfies the criterion. The presented algorithm can thus be used for any coverage criteria. To specify coverage criteria we use observer automata with parameters that is used to monitor and accept traces. Paper II and Paper IV address the research question:

How can model-checking techniques for timed systems be specialized for test-suite generation to use less time and memory?

In Paper I we had to change the model for each coverage criterion. This is automated in Paper II where we also present a pruning technique. In Paper IV we present a novel algorithm for test case generation. Our experiments show that the algorithm outperforms (in time and memory) our earlier algorithms that are based on reachability analysis. We address the last research question in Paper V and Paper VI:

Is it possible to apply model-based specification and generation techniques to test an industrial-sized timed system?

In these papers we describe the novel test case generation tool CO \checkmark ER and the framework around it. In Paper V we describe a case study done in cooperation with Ericsson AB. In the case study, we automate test case generation and ex-

ecution for a WAP gateway developed by Ericsson. We execute and evaluate test suites generated from a model of the gateway, according to several coverage criteria. The coverage criteria were specified with coverage observers. The used model consist of multiple layers with simultaneous transitions. We now give a summary of the enclosed papers follows.

Paper I: Time-optimal Real-Time Test Case Generation using Uppaal

In Paper I we demonstrate how the problem of test case selection can be transformed to a reachability problem. We show how test suites for both single test purposes and coverage criteria can be generated with the UPPAAL model checking tool. Especially, we apply the minimum cost reachability analysis, found in UPPAAL, to generate a test suite that has the shortest execution time, and still fulfills the given coverage criteria. The rationale for this is not primary to stress the system but not to wait unnecessary long time when executing the test cases. In this paper we also define the deterministic input enabled output urgent timed automata (DIEOU-TA).

We annotate the model to keep track of coverage items. The coverage is saved in auxiliary variables in the model. If a state is found during state-space exploration, where all such variables are set, then the trace to that state is a trace that fulfills the coverage criterion. Thus, the problem of finding a trace that fulfills a coverage criterion is reduced to a reachability problem. If it is impossible to find full coverage with only one test case, we show how to decorate the model so that the system can take a transition to its initial state with the accumulated coverage item information kept intact.

We present experiments on how the technique scales regarding time and memory of the test case generation. A major drawback of this approach is that the model must be annotated for every new coverage criterion. In Paper II we describe an implementation that makes such modifications superfluous.

Paper II: A Test Case Generation Algorithm for Real-Time Systems

In Paper II we automate the test case generation of the coverage criteria described in Paper I. We present an abstract algorithm for symbolic reachability analysis with coverage, in which a pruning technique is used. We also describe some aspects of a prototype implementation and show some experiments, which demonstrate the benefits of the pruning technique.

In this paper we do not use manually annotated auxiliary variables to store the information of the coverage items as in Paper I. Instead we keep track of the information by extending the ordinary model states with additional data.

Bit-vectors are used to represent both the covered items and (as in the case of the definition use pair data-flow criteria) information, which affects the future possibilities to reach cover items.

Our pruning technique can be defined as follows: A state s' is pruned if the algorithm has, before exploring s' , found a state s such that s and s' have equal model states and the coverage of s' is subsumed by the coverage of s . As we use a symbolic representation of time we also require the symbolic model state of s' to be included in the symbolic model state of s to prune s' .

A weakness of the solution in Paper II is that only the criteria that are implemented in the tool are available to the user. This issue is addressed in Paper III. Another issue is the efficiency of the algorithm, which is addressed in Paper IV.

Paper III: Specifying and Generating Test Cases Using Observer Automata

In Paper III we present a technique for specifying coverage criteria and a method for generating test suites for systems whose behaviors can be described as extended finite state machines (EFSM). The technique is demonstrated for EFSMs but is also applicable for other types of models, e.g., timed automata (DIEOU-TA). We use observer automata to monitor traces. The technique is expressive and we demonstrate this by specifying a number of well-known coverage criteria based on control- and data-flow information using observer automata. Further we represent the set of observer locations in a state as a bit-vector, and show how to encode the transition from one set of observer locations to another, given the EFSM transition.

The coverage observer language has its most obvious benefits with data-flow coverage criteria. They can also be used to express equivalence-class based coverage criteria as projections of the state-space of the EFSM.

The paper describes a language to specify predicates and some example predicates, which are useful in EFSM models. These are used to show the principles of observers and their semantics. The predicates used in an implementation of the observer technique will of course depend on the context.

Paper III is a paper about ideas of how to express coverage and how to monitor coverage during model exploration. To evaluate if our approach is useful in practice we have implemented the ideas in our tool CO \checkmark ER that is presented in Paper VI and used in a case study described in Paper V.

Paper IV: A Global Coverage Algorithm for Model Based Testing

In Paper IV we present a novel reachability algorithm for test case generation that makes use of the union of the coverage found in every search branch throughout the exploration.

The algorithm explores a state only if it might increase the total coverage. Every time a new (fully satisfied) coverage item is found, the algorithm puts the trace into a preliminary return set. The algorithm exits its first phase when the total coverage has reached its threshold or if there is no more states to explore. The preliminary return set is reduced in a second phase. We use a simple algorithm to reduce the preliminary return set. The set of traces is reduced until there is at least one unique coverage item covered in each trace. The algorithm then returns the reduced set of traces.

When the number of coverage items is known in advance or is specified by a user the algorithm returns a set of traces that together cover the required number of coverage items. As we use a breadth-first search strategy we find each coverage item in a minimal number of steps. Thus the search depth is limited to the coverage item that needs the most number of steps to be satisfied. When on the other hand the number of coverage items is unknown the algorithm will return after exploring a smaller state-space than the compared algorithm from Paper II as satisfied coverage items do not need to be considered when two states are compared.

Compared with algorithms that first generate the state space of the model and then compute test cases, our algorithm benefits from techniques developed for symbolic model checking, e.g., inclusion of symbolic states in timed automata models.

In the paper we provide performance data from runs of trace generations to compare our novel algorithm with the one in Paper II. In a number of experiments our new algorithm is able to generate test suites for bigger models and for more complex coverage criteria than the one in Paper II.

Paper V: Model-Based Testing of a WAP Gateway: an Industrial Case Study

In Paper V we present experiences from a case study where the techniques from Paper I to IV have been applied. In the case study we verify that a *wireless application protocol* (WAP) gateway conforms to its specification. The WAP gateway is developed by Ericsson and used in mobile telephone networks to connect mobile phones with the Internet.

We present a complete test bed including generation and execution of test cases. The test bed takes as input a model expressed as a timed automata

network and a coverage criterion expressed as an observer, and returns a set of test cases with their verdicts.

The CO \checkmark ER tool is used to produce abstract test cases in the form of timed traces. A custom-written tool, tr2mac [Vil05] is used to create scripts from the timed traces produced by CO \checkmark ER. It identifies the packages sent between the environment and the gateway in the traces and outputs a script file that includes instructions to send/receive packages together with the delays between them. The package names refer to prefabricated packages in a repository. The Ericsson legacy tool TSC2 that is able to evaluate the timing and data of the received packages executes the scripts, produced by tr2mac.

The functionality of two layers, the *session layer* (WSP) and the *transaction layer* (WTP), have been modeled in great detail. Other layers have been abstracted as much as possible. The modeling of the interaction between the gateway and its environment is done so that all data relevant for the WSP and WTP layers is represented in the model.

In the WTP layer, sequence numbers called *transaction ids* (TIDs) are used to separate the transactions. The domain of the TIDs is large so it would not be feasible to exhaust the state space without abstraction. Unfortunately, the behavior of the system is dependent on the relations of the TID values.

With help of an abstraction that preserve the relations between the TID values present in a model state, we are able to make equivalence classes of states. The classes are equivalent with respect to future coverage, as long as the TID values are not part of the coverage criterion themselves. A comparison between two states is done in the abstract state space, but the transitions are computed with the concrete TID values to simplify the generation of real test cases.

The case study was successful. Traces for test suites were produced by CO \checkmark ER for a handful coverage criteria. The traces were successfully converted to test scripts that were executed against the WAP gateway. Discrepancies were found between the model and the real system. Analysis of the discrepancies gave that some errors were located to the model and some were located to the system.

Paper VI: CoVer – A Test Case Generation Tool for Real-Time Systems

In this paper we present the CO \checkmark ER tool in which we have implemented the ideas from Paper I to V. The paper briefly describes some of the features of the tool including: (i) an efficient generation algorithm (Paper IV), (ii) an observer automaton input language for specification of the coverage criteria (Paper III), (iii) model compatibility with UPPAAL, (iv) generation of test suites in XML format with symbolic names, and (v) a query language that

can further specify how test case generation should be performed, e.g., which automata to consider for coverage.

The main design decisions of the tool are presented in the paper. For instance we decided that the observers should be model independent and that the query language should be able to restrict the possible values of the observer parameters.

The current implementation is based on the verification engine of the UP-PAAL tool called *verifyta*. One of the parts that are added in *COVER* is the observer engine library. The library is designed to deliver a coverage observer service to any model checking tool that adapts to its interface. The paper presents on an architectural level how new parts are integrated with *verifyta*.

6. Related Work

6.1 Model-Based Testing of Timed Systems

In the literature, there are several work of test case generation from specifications of timed systems [SVD01, CKL98, ENDKE98, CO00, Kho02, NS03, LMN05].

To make the automata deterministic Springintveld et al [SVD01] use similar restrictions as we do in Paper I. For testing purposes, they discretize the time into a grid automata (with constant delays) based on regions and generate test cases using Chow's classical W-algorithm [Cho78] based on characterization sets. Another work that defines a determinizable timed automata is by Khomsi et al. [KJM04].

En-Nouaary et al. [ENDKE98] extend the generalized partial W-method (Wp-method) [LvBP94] to timed input/output automata. In a grid-automaton with synchronized actions all actions are discrete including delays, as in an FSM. The automata used have no restrictions as our automata and if we consider input and delay as tester actions and output and clock reset as response actions such automaton is non-deterministic. En-Nouaary et al. transform the automaton to a non-deterministic automaton with transitions labeled with tester action/response action pairs. This is an observable non-determinism on which the generalized Wp-method can be applied. After the last transition the current state can be calculated by the observed response. We do not use discrete states in the model we are generating from, thus our state space is not sensitive to the size of the delays in the same manner. For number of clocks used, the complexity increase roughly equal for the both approaches.

Typical for methods using the W-method or other checking sequence techniques is that they rely on a fault hypothesis. Typically the faults can be; action or output faults, transfer fault, extra transition in implementation, and missing transition in the implementation. The method is sensitive for the number of states in the specification. In our method we can often find test cases without even generating all states in the specification.

It can be helpful to give restrictions of the environment to avoid generating uninteresting test cases. These restrictions can also be seen as guiding to especially wanted test cases, e.g., it can be of the form of a test purpose. Castanet et al. [CKL98] make a synchronous product between a system automaton and an acyclic test purpose automaton, where the test purpose automaton has an accepting subset. If this subset can be reached then a test of the given purpose can be generated. We show in Paper I that this approach is valid in our setting.

Cardell-Oliver [CO00] uses UPPAAL timed automata as we do. Checking sequences are not used for the full system but for some aspects. Test views that are part of a test plan are used to focus on some functionality, which make the size of the system suitable for generating test cases. Test views are related to test purposes.

Khoumsi [Kho02] uses timed automata, assuming determinism but not output urgency. The model is transformed into an *se-FSA*, which is an automaton where clocks and their operations have been replaced with *set* and *expire* actions. The translation keeps the behavior of the automata. This is equal to the symbolic representation in UPPAAL with the extension of equivalence classes done by Nielsen and Skou [NS03] to get feasible traces. A test execution program can get exact instructions from the *se-FSA* for the needed timers and the values that must be set, the output to be observed, and the inputs possible to send. The expiring of the timers and the output gives the test execution program information of the state changes the implementation should do in order to conform to the specification. Test sequences are generated from the *se-FSA* with the generalized Wp-method.

Another type of restricted non-determinism is event recording automata (ERA) [AFH94], which is a determinizable subset of timed automata. ERA is basically timed automata where there is one clock per action and a clock is reset on the corresponding action. ERA is used in the work of Nielsen and Skou [NS03] where the model is determinized and the symbolic states found is divided into equivalence classes. In test cases where the specification gives the implementation freedom to choose an output (among several) the outcome trace is classed as a *may* trace. If it does not give the freedom then the trace is a *must* trace. May and must traceability was proposed by De Nicola and Hennessy [NH84]. The base of the symbolic state exploration of Nielsen and Skou's tool is UPPAAL, as in our tool. We do not require ERA, but deterministic automata. In fact, as we require deterministic automata we always know at what timepoints clocks are reset in the system. The graph constructed by Nielsen and Skou is general and could be used as any untimed graph to find test cases. In contrast, the state-space generated by our reachability algorithm is sensitive to the coverage criterion given to our algorithm.

Higashino et al. [HNTC99] use timed I/O automata where the timing of the specified system is not controllable. The intervals where the action can happen are analyzed and may and must test are generated with the UIOv-method where, for each state, a transfer sequence and a succeeding unique input/output sequence are treated as a test case.

Cleaveland and Zwarico [CZ91] present a framework for generating preorders not only for non-determinism but also for timing behavior. A system may (or must) be "faster-than" a compared system. This is for example useful when stimuli is only accepted for a certain amount of time because no output is given when the system stops accepting the input. Krichen and Tripakis [KT04] use non-deterministic and partially observable timed automata. They

also analyze must and may preorder of trace inclusion. Further they generate digital clock-tests which measure time with a periodic clock. This is useful for a test program in practice and similar to the approach by Khoumsi [Kho02]. The method can simulate clock drifts which can generate less strict test cases. The method can be used both offline (before test case execution) and online (during test case execution).

Larsen et al. [LMN05] is also using online testing. The tool used is T-UPPAAL now renamed to UPPAAL TRON. As our tool, UPPAAL TRON is based on UPPAAL but has no restrictions to deterministic automata as our tool has. TRON tracks possible states of the specification during execution and chooses input randomly. If there is no possible state an error is reported. There is not yet any guidance in TRON, so no specific coverage is guaranteed.

6.2 Test Case Generation with Coverage Criteria

Our coverage observers are related to the work of Mandrioli et al. [MMM95] that uses specification written in the TRIO language that extends classical temporal logic to deal explicitly with time measures. In the work test cases are generated using a history generator and a history checker. Håkansson et al. [HJL03] use TRIO to generate a test oracle. The explicit purpose of the test oracles is to check safety properties during test execution.

Temporal logic is also used in the work of Hong et al [HLSU02, HCL⁺03] to describe data-flow coverage criteria. They use a model checker to generate test cases for a CTL formula. The implicit for-all quantification in our work is a novelty. Hong et al. have to make a reachability search for each coverage item. The witness returned is a trace which covers that particular coverage item.

Related are also the work of Friedman et al. [FHNS02]. The parameters of our observers can be used in ways similar to projection coverage. To extract the location from a state or the edge used to make a transition is a projection of the state space. Friedman et al. use a test generation tool GOTCHA. Various projections on the state graph of an EFSM can be specified with a simple programming interface provided by GOTCHA.

There are some work that uses model checking to generate test suites that satisfies a coverage criterion. Rayadurgam and Heimdahl [RH01] generate test cases with the SMV model checker. By hard coded coverage items they show that a model checker can generate tricky coverage criteria as MC/DC. The SPIN model checker has been used by Gargantini et al. [GRR03] for similar purposes.

6.3 Tools for Model-Based Testing

There is a wide variety of test tool both in academia and in industry apart from the already mentioned tools. The STG tool [CJRZ02] is a symbolic test generation tool using the IOSTS (Input Output Symbolic Transition System) [RdBJ00]. Rusu et al. generates test cases from test purposes [RdBJ00] this is similar to our test purposes generated in Paper I. The STG tool is related to TorX [dBRS⁺00] developed by du Bousquet et al. STG and TorX are conformance test tools based on the ioco [Tre96, TdV00] conformance relation defined by Tretmanns. Bouquet and Legeard describes the BZ-Testing-Tools [BL03] – a tool-set for animations and test generation from B, Z, and state-chart specifications.

Other more loosely related tools are tools for model checking of code. The Bandera Tool Set [HD01] model checks properties of concurrent java software. Bandera compiles java code and a specification in BSL (Bandera Specification Language) to the input language of several model checkers. Bandera uses program slicing and data-abstraction (abstract interpretation) on the model driven by the specification when it transform the code to the model checking languages. BLAST [BCH⁺04] is a verification tool for checking temporal safety properties of C programs. It constructs an abstract reachability tree with program locations and truth values of predicates. It has two levels of specification languages, i.e., observer automata and relational queries. Observer automata monitors safety properties and relational queries that may specify both structural and semantic properties much as our coverage observers although not used for the same purpose. Other tools that model check programs are SLAM [BR01] by Microsoft and JFP [VHB⁺03, KPV03] by NASA.

Random testing addresses the problem of huge number of data sets. Specialized tools for random testing includes DART [GKS05] by Godefroid et al. GAST [KATP03] by Koopman et al., and QuickCheck [CH00] by Claessen and Huges.

The GOTCHA tool [BGH⁺99] by Benjamin et al., mentioned in the previous section is a coverage-driven tool for generating test cases for hardware architectures. GOTCHA was implemented as an extension of the Mur Φ model checker. It has its own specification language GDL and builds a C++ file containing the test case generation algorithm and the final state machine model.

The TGV tool [JJ05] is another ioco-based tool that is very rich in input languages, e.g., SDL, UML, and IF. It builds a synchronous products of the model and a test purpose on-the-fly to generate a test case for the purpose. An interesting feature is that it defines *points of control and observation* (PCO) that can be seen as multiple environment views similar to our approach in the case study in Paper V of a WAP [For01] gateway.

7. Conclusion and Future Work

7.1 Conclusion

Our objectives with this thesis work have been to develop test techniques for real-time systems, implement them in a tool, and evaluate them in a case study. We have developed the tool CO \checkmark ER that is able to produce test suites. The user specifies a model of a system and a coverage criterion from which CO \checkmark ER produces a test suite that satisfies the coverage criteria. The development of CO \checkmark ER began with the model-checker UPPAAL. By adding feature by feature we have turned the model checker into a test generation tool. This evolution is described in the thesis.

We first show that reachability algorithms are useful for test case generation of timed systems. Especially we generate test suites with full coverage and minimal cost. We continue with the problem of specifying coverage criteria and present an observer automata language for this purpose. The language is flexible and can express many different coverage criteria including logical, data-flow, and state projection criteria. The user defines her own coverage criterion that at the same time indirectly defines the way the test case generation algorithm works in the tool. Further, we develop a specialized reachability algorithm for test case generation that avoids unnecessary state-space explosion, caused by the coverage criterion, by using global information of the exploration. All ideas presented in the thesis have been implemented in the CO \checkmark ER tool. Except for proving the ideas feasibility we have also been able to evaluate their efficiency. In an industrial-sized case study CO \checkmark ER has been successfully used to generate test cases from a model of a WAP gateway. For the case study we have developed a test bed that fully automates generation and execution of test cases, and decides their verdicts.

The experiences from the case study clearly show that the problem of finding a good abstraction for the system model is the hardest problem for a test engineer that use model-based testing. Tests generated from too abstract models may not test the system enough and too detailed models can be as hard to produce as the real system. For a model that has too much details, shortage of time and memory can make it impossible to systematically explore the state-space needed to find test cases that satisfies a coverage criterion. In our case study, several sequence numbers for the WTP protocol were needed to test all functionality. We let the numbers be in the model but used an abstraction technique to explore only an abstract state space.

7.2 Future Work

It would be very interesting to investigate how the observer technique can be used for other types of models than the EFSMs and timed automata we have used. We have implemented the tool so that the computations of coverage is in a separate library with a generic interface. With an adaptor (that is also used for UPPAAL today) other model checkers or similar state exploration tools could make use of the library. We believe that the high abstraction level of the predicates (guards of the observer edges) in the language makes it possible to establish standard descriptions for coverage criteria, i.e., observer definitions can be reused for many modeling languages.

One type of coverage that is currently not possible in CO \checkmark ER is coverage on boundary values of clocks. This is partly because of the design decision that we in the current implementation separate the transitions in the model that generates new model states and the computation of the new coverage associated with the new states. Such separation is always possible for explicit states but not for symbolic states. For a symbolic state the included transitions can have different coverage, which is the case in e.g., boundary values. In CO \checkmark ER time is treated symbolically and thus coverage criteria with predicates that include clock evaluations can be satisfied for a part of a symbolic transition. In such case a generated symbolic target state would need to be split in parts.

In the current version of CO \checkmark ER partial satisfied coverage items are defined by observer locations with parameters. For each such location the parameter is a tuple of fixed length. Coverage criteria as, e.g., all-paths (for any finite length of the paths) can be covered if a list of dynamic length is used as a parameter. This is not implemented in the current tool because of the potential performance penalty. Another reason is the we can not guarantee termination if dynamic length lists are used as observer parameters, which we can in the current implementation.

In the future the CO \checkmark ER tool could be integrated into a Graphical User Interface. Support for graphical observer automata editing and ability to manage a set of traces (the generated test suite) are examples of how the tool can become more available for non-expert users. In UPPAAL it is now possible to include routines written in C-code that are called when an edge is exercised. Thus, possible future work includes development of coverage for the C-code in CO \checkmark ER.

Visualization by modeling the functionality of system is a best practice in industry, advocated by, e.g., XP [BF00, hw06] and RUP [Tea06, Str01]. In the Spring Web Flow [LDDY06] automata are used to define how the user can interact with a web site. The automata are part of the code and not (only) for documentation. This is an example of high level modeling with automata that can be addressed by test case generation tools in the future. The techniques presented in this thesis can be helpful to generate test cases for existing test tools, e.g. QTP, Winrunner by Mercury. CO \checkmark ER has been applied in a case

study for telecom protocols, but new and unexpected application areas may appear for model-based testing tools.

References

- [ABC82] W. Richards Adrion, Martha A. Branstad, and John C. Cherniavsky. Validation, verification, and testing of computer software. *ACM Comput. Surv.*, 14(2):159–192, 1982.
- [ABM98] P. E. Ammann, P. E. Black, and W. Majurski. Using model checking to generate tests from specifications. In *Formal Engineering Methods, 1998. Proceedings. Second International Conference on*, pages 46–54. IEEE, 1998.
- [AD94] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.
- [AFH94] R. Alur, L. Fix, and T. A. Henzinger. Event-clock automata: a determinizable class of timed automata. In *Proc. 6th Int. Conf. on Computer Aided Verification*, volume 818 of *Lecture Notes in Computer Science*. Springer–Verlag, 1994.
- [BAKD01] C. Bourhfir, E. Aboulhamid, F. Khendek, and R. Dssouli. Test cases selection from sdl specifications. 35(6):693–708, May 2001.
- [BCH⁺04] D. Beyer, A. J. Chlipala, T. A. Henzinger, R. Jhala, and R. Majumdar. The blast query language for software verification. In R. Giacobazzi, editor, *Proc. 11th International Static Analysis Symposium (SAS 2004)*, volume 3148 of *Lecture Notes in Computer Science*, pages 2–18. Springer–Verlag, 2004.
- [BF00] K. Beck and M. Fowler. *Planning Extreme Programming*. Addison-Wesley Professional, 2000.
- [BGH⁺99] M. Benjamin, D. Geist, A. Hartman, Y. Wolfsthal, G. Mas, and R. Smeets. A study in coverage-driven test generation. In *Design Automation Conference, 1999. Proceedings. 36th*, pages 970–975. IEEE Computer Society, 1999.
- [BL03] F. Bouquet and B. Legnard. Reification of executable test scripts in formal specification-based test generation: The java card transaction mechanism case study. In *FME 2003*, volume 2805 of *Lecture Notes in Computer Science*, pages 778–795. Springer–Verlag, 2003.

- [BR01] T. Ball and S.K. Rajamani. Automatically validating temporal safety properties of interfaces. In M.B. Dwyer, editor, *Proc. 8th International SPIN Workshop on Model Checking Software (SPIN'01)*, volume 2057 of *Lecture Notes in Computer Science*, pages 103–122. Springer–Verlag, 2001.
- [CH00] Koen Claessen and John Hughes. Quickcheck: A lightweight tool for random testing of haskell programs. In *In Proc. of International Conference on Functional Programming (ICFP), ACM SIGPLAN*, 2000.
- [Cho78] T.S. Chow. Testing software design modeled by finite-state machines. *IEEE Trans. on Software Engineering*, 4(3):178–187, 1978.
- [CJRZ02] D. Clarke, T. Jérón, V. Rusu, and E. Zinovieva. Stg: A symbolic test generation tool. In J.-P. Katoen and P. Stevens, editors, *Tools and Algorithms for the Construction and Analysis of Systems : 8th International Conference, (TACAS'02)*, volume 2280 of *Lecture Notes in Computer Science*, pages 324–356. Springer–Verlag, 2002.
- [CKL98] R. Castanet, O. Koné, and P. Laurençot. On The Fly Test Generation for Real-Time Protocols. In *International Conference in Computer Communications and Networks*, Lafayette, Louisiana, USA, October 12-15 1998. IEEE Computer Society Press.
- [CM94] J.J. Chilenski and S.P. Miller. Applicability of modified condition/decision coverage to software testing. *Software Engineering Journal*, 9(5):193–200, Sept. 1994.
- [CO00] R. Cardell-Oliver. Conformance Testing of Real-Time Systems with Timed Automata. *Formal Aspects of Computing*, 12(5):350–371, 2000.
- [CPRZ89] L. A. Clarke, A. Podgurski, D. J. Richardsson, and S. J. Zeil. A formal evaluation of data flow path selection criteria. *IEEE Trans. on Software Engineering*, SE-15(11):1318–1332, November 1989.
- [CZ91] R. Cleaveland and A. E. Zwarico. A theory of testing for real-time. In *LICS*, pages 110–119, 1991.
- [dBRS⁺00] L. du Bousquet, S. Ramangalahy, S. Simon, C. Viho, A. Belinfante, and R.G. de Vries. Formal test automation: The conference protocol with tgv/torx. In H. Ural, R.L. Probert, and G. von Bochmann, editors, *IFIP 13th Int. Conference on Testing of Communicating Systems (TestCom 2000)*. Kluwer Academic Publishers, 2000.
- [ENDKE98] A. En-Nouaary, R. Dssouli, F. Khendek, and A. Elqortobi. Timed Test Cases Generation Based on State Characterization Technique. In 19th

IEEE Real-Time Systems Symposium (RTSS'98), pages 220–229, December 2–4 1998.

- [FHNS02] G. Friedman, A. Hartman, K. Nagin, and T. Shiran. Projected state machine coverage for software testing. In *Proc. ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 134–143, 2002.
- [For01] WAP Forum. Wireless transaction protocol, version 10-jul-2001. on-line, 2001. <http://www.wapforum.org/>.
- [GKS05] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: directed automated random testing. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 213–223, New York, NY, USA, 2005. ACM Press.
- [GRR03] A. Gargantini, E. Riccobene, and S. Rinzivillo. Using spin to generate tests from asm specifications. In *Abstract State Machines 2003. Advances in Theory and Practice: 10th International Workshop, ASM 2003, Taormina, Italy, March 3-7, 2003. Proceedings*, volume 2589 of *Lecture Notes in Computer Science*, page 263. Springer–Verlag, 2003.
- [HCL⁺03] H.S. Hong, S.D. Cha, I. Lee, O. Sokolsky, and H. Ural. Data flow testing as model checking. In *ICSE'03: 25th Int. Conf. on Software Engineering*, pages 232–242, May 2003.
- [HD01] John Hatcliff and Matthew Dwyer. Using the bandera tool set to model-check properties of concurrent java software. In J.-P. Katoen and P. Stevens, editors, *CONCUR 2001 - Concurrency Theory, 12th International Conference, Aalborg, Denmark, August 20-25, 2001, Proceedings*, volume 2154 of *Lecture Notes in Computer Science*, pages 39–58. Springer–Verlag, 2001.
- [Her76] P.M. Herman. A data flow analysis approach to program testing. *Australian Computer J.*, 8(3), 1976.
- [HJL03] J. Håkansson, B. Jonsson, and O. Lundqvist. Generating on-line test oracles from temporal logic specifications. *Int. Journal on Software Tools for Technology Transfer*, 4(4):456–471, Sept. 2003.
- [HLSU02] H.S. Hong, I. Lee, O. Sokolsky, and H. Ural. A temporal logic based theory of test coverage. In J.-P. Katoen and P. Stevens, editors, *Tools and Algorithms for the Construction and Analysis of Systems : 8th International Conference, (TACAS'02)*, volume 2280 of *Lecture Notes in Computer Science*, pages 327–341. Springer–Verlag, 2002.

- [HNTC99] T. Higashino, A. Nakata, K. Taniguchi, and A. R. Cavalli. Generating Test Cases for a Timed I/O Automaton Model. In G. Csopaki, S. Dibuz, and K. Tarnay, editors, *Testing of Communicating Systems: Method and Applications, IFIP TC6 12th International Workshop on Testing Communicating Systems (IWTCS), September 1-3, 1999, Budapest, Hungary*, volume 147 of *IFIP Conference Proceedings*, pages 197–214. Kluwer, 1999.
- [Hol97] G.J. Holzmann. The model checker SPIN. *IEEE Trans. on Software Engineering*, SE-23(5):279–295, May 1997.
- [hw06] <http://www.extremeprogramming.org/> website. Extreme programming: A gentle introduction. online, 2001–2006. <http://www.extremeprogramming.org/>.
- [JJ05] C. Jard and T. Jéron. Tgv: theory, principles and algorithms a tool for the automatic synthesis of conformance test cases for non-deterministic reactive systems. *Int. Journal on Software Tools for Technology Transfer*, 7(4):293–296, August 2005.
- [KATP03] P. Koopman, A. Alimarine, J. Tretmans, and R. Plasmeijer. Gast: Generic automated software testing. In *Implementation of Functional Languages, 14th International Workshop, IFL 2002 Madrid, Spain, September 16-18, 2002 Revised Selected Papers*, volume 2670 of *Lecture Notes in Computer Science*, pages 84–100. Springer–Verlag, 2003.
- [Kho02] A. Khoumsi. A method for testing the conformance of real-time systems. In W. Damm and E-R. Olderog, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems: 7th International Symposium, FTRTFT 2002*, volume 2469 of *LNCS*. Springer–Verlag, September 2002.
- [KJM04] A. Khoumsi, T. Jéron, and H. Marchand. Test case generation for non-deterministic real-time systems. In A. Petrenko and A. Ulrich, editors, *Proc. 3rd International Workshop on Formal Approaches to Testing of Software 2003 (FATES’03)*, volume 2931 of *Lecture Notes in Computer Science*, pages 131–151. Springer–Verlag, 2004.
- [KPV03] S. Khurshid, C. S. Pasareanu, and W. Visser. Generalized symbolic execution for model checking and testing. In H. Garavel and J. Hatcliff, editors, *Tools and Algorithms for the Construction and Analysis of Systems: 9th International Conference, TACAS 2003, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland, April 7-11, 2003. Proceedings*, volume 2619 of *Lecture Notes in Computer Science*, pages 553–568. Springer–Verlag, 2003.

- [KT04] M. Krichen and S. Tripakis. Black-box conformance testing for real-time systems. In Laurent Mounier Susanne Graf, editor, *Proc. 11th International SPIN Workshop on Model Checking Software (SPIN'04)*, volume 2989 of *Lecture Notes in Computer Science*, pages 182–197. Springer–Verlag, 2004.
- [LDDY06] S. Ladd, D. Davison, S. Devijver, and C. Yates. *Spring MVC and Web Flow*. APress, 2006.
- [LK83] J. W. Laski and B. Korel. A data flow oriented program testing strategy. *IEEE Trans. on Software Engineering*, SE-9(3):347–354, May 1983.
- [LMN05] K. G. Larsen, M. Mikucionis, and B. Nielsen. Online testing of real-time systems using uppaal. In J. Gabowski and B. Nielsen, editors, *Proc. 4th International Workshop on Formal Approaches to Testing of Software 2004 (FATES'04)*, volume 3395 of *Lecture Notes in Computer Science*, pages 79–94. Springer–Verlag, 2005.
- [LPU02] B. Legeard, F. Peureux, and M. Utting. Automated boundary testing from z and b. In L.-H. Eriksson and P. Lindsay, editors, *FME 2002: Formal Methods - Getting IT Right : International Symposium of Formal Methods Europe, Copenhagen, Denmark, July 22-24, 2002. Proceedings*, volume 2391 of *Lecture Notes in Computer Science*, page 263. Springer–Verlag, 2002.
- [LPY97] K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a Nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, October 1997.
- [LvBP94] G. Luo, G. von Bochmann, and A. Petrenko. Test selection based on communication nondeterministic finite state machines using and generalised wp-method. *IEEE Trans. on Software Engineering*, SE-20(2):149–162, 1994.
- [LY96] D. Lee and M. Yannakakis. Principles and Methods of Testing Finite State Machines—A Survey. *Proceedings of the IEEE*, 84(8):1090–1123, August 1996.
- [MMM95] D. Mandrioli, S. Morasca, and A. Morzenti. Generating Test Cases for Real-Time Systems from Logic Specifications. *ACM Transactions on Computer Systems*, 13(4):365–398, 1995.
- [Mye79] G. Myers. *The Art of Software Testing*. Wiley-Interscience, 1979.
- [NH84] R. De Nicola and M. C. B. Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 34(1-2):83–133, 1984.

- [NS03] Brian Nielsen and Arne Skou. Automated test generation from timed automata. *International Journal on Software Tools for Technology Transfer*, 5:59–77, 2003.
- [Nta88] S. Ntafos. A comparison of some structural testing strategies. *IEEE Trans. on Software Engineering*, 14:868–874, 1988.
- [RCT92] RCTA, Washington D.C., USA. *RTCA/DO-178B, Software Considerations in Airbourne Systems and Equipment Certifications*, December 1992.
- [RdBJ00] V. Rusu, L. du Bousquet, and T. Jérón. An approach to symbolic test generation. In *Int. Conf. on Integrating Formal Methods*, volume 1945 of *Lecture Notes in Computer Science*, pages 338–357. Springer–Verlag, 2000.
- [RH01] S. Rayadurgam and M. P. E. Heimdahl. Test-sequence generation from formal requirement models. *hase*, 00:0023, 2001.
- [Rou06] Vlad Roubtsov. Emma reference manual. online, 2006. <http://emma.sourceforge.net/reference/reference.pdf>.
- [RW85] S. Rapps and E. J. Weyuker. Selecting software test data using data flow information. *IEEE Trans. Softw. Eng.*, 11(4):367–375, 1985.
- [Str01] L. Strand. Docendo, 2001.
- [SVD01] J. Springintveld, F. Vaandrager, and P.R. D’Argenio. Testing Timed Automata. *Theoretical Computer Science*, 254(1-2):225–257, March 2001.
- [TdV00] J. Tretmans and René G. de Vries. On-the-fly conformance testing using spin. *Int. Journal on Software Tools for Technology Transfer*, 2(4):282–292, 2000.
- [Tea06] Rational Team. A rational white paper: The rational unified process for systems engineering. online, 2006. <ftp://ftp.software.ibm.com/software/rational/web/whitepapers/2003/TP165.pdf>.
- [Tre96] J. Tretmans. Test generation with inputs, outputs, and quiescence. In T. Margaria and B. Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems: 2nd Int. Workshop (TACAS’96)*, volume 1055 of *Lecture Notes in Computer Science*, pages 127–146. Springer–Verlag, 1996.
- [VHB⁺03] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engineering Journal*, 10(2), April 2003.

- [vHW03] W. von Hagen and K. Wall. *The Definitive Guide to GCC*. Apress, 2003.
- [Vil05] Per Vilhelmsson. A test case translation tool - from abstract test sequences to concrete test programs. Technical report, Department of Information Technology, Uppsala University, 2005.