IT Licentiate theses
2006-002

Model-Based Test Case Selection and Generation for Real-Time Systems

ANDERS HESSEL

UPPSALA UNIVERSITY
Department of Information Technology

UPPSALA
UNIVERSITET

Model-Based Test Case Selection and Generation for Real-Time Systems

BY
ANDERS HESSEL

March 2006

DIVISION OF COMPUTER SYSTEMS
DEPARTMENT OF INFORMATION TECHNOLOGY
UPPSALA UNIVERSITY
UPPSALA
SWEDEN

Dissertation for the degree of Licentiate of Philosophy in Computer Science
at Uppsala University 2006

Model-Based Test Case Selection and Generation for Real-Time Systems

Anders Hessel
Anders.Hessel@it.uu.se

Division of Computer Systems
Department of Information Technology
Uppsala University
Box 337
SE-751 05 Uppsala
Sweden

http://www.it.uu.se/

ISSN 1404-5117
Printed by the Department of Information Technology, Uppsala University, Sweden

Abstract

Testing is the dominating verification technique used in industry today, and many man-hours and resources are invested in the testing of software products. To cut down the cost of testing, automated test execution becomes more and more popular. However, the selection of which tests to be executed is still mainly a manual process that is error prone, and often without sufficient guarantees that the system will be systematically tested. A way to achieve systematic testing is to ensure that the tests satisfy a required coverage criterion.

In this thesis two main problems are addressed: the problem of how to formally specify coverage criteria, and the problem of how to generate a test suite from a formal system model, such that the test suite satisfies a given coverage criterion. We also address the problem of how to generate an optimal test suite, e.g., with respect to the total time required to execute the test suite.

Our approach is to convert the test case generation problem into a reachability problem. We observe that a coverage criterion consists of a set of items, which we call coverage items. The problem of generating a test case for each coverage item can be treated as separate reachability problems. We use an on-the-fly method, where coverage information is added to the states of the analyzed system model, to solve the reachability problem of a coverage item. The coverage information is used to select a set of test cases that together satisfy all the coverage items, and thus the full coverage criterion.

Based on the 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 in the literature. Two different ways to generate test suites form a system model and a given coverage criterion are presented. The first way is based on model annotations and uses the model checker Uppaal. The second way, where no annotations are needed, is a modified reachability analysis algorithm that is implemented in an extended version of the Uppaal tool.

Acknowledgments

I would like to thank my supervisor Paul Pettersson for supervising, co-authoring, and proofreading my work. There have been a lot of suggestions for improvements. Paul has been involved in all my research activities. As Paul is active in both the testing group and the UPPAAL group, I have had the opportunity to learn a lot from both communities.

I would also like to thank my co-supervisor Bengt Jonsson, who has given me important feedback on the structure of the thesis. Bengt is co-author of Paper C, and a brilliant researcher that I am very glad to work with. Bengt was my (main) supervisor for the first year, and helped me through my master thesis report, which gave me my first scientific writing lessons. I am also thankful for the proofreading done by Johan Blom and John Håkansson.

Besides the supervisors I have had a very fruitful collaboration with my other co-authors; Johan Blom (Paper C), Kim G. Larsen, Brian Nielsen, and Arne Skou (Paper A) and the other members of the testing group, Olga Grinstein and Therese Berg.

I would specially like to thank the people in the UPPAAL group. Apart form the already mentioned persons I would also like to mention Wang Yi, Alexander David, Johan Bengtsson, Leonid Mokrushin, Gerd Behrmann, and John Håkansson. It has been a lot of coffee, especially with John. I also have had a lot of fun but that is another story.

Last but not least my fiancé Anna, just for being you.

List of Enclosed Papers

Paper A:

Anders Hessel, Kim G. Larsen, Brian Nielsen, Paul Pettersson, and Arne Skou. Time-optimal Real-Time Test Case Generation using UPPAAL. In Alexandre Petrenko and Andreas Ulrich, editors, *Proceedings of the 3*rd *International Workshop on Formal Approaches to Testing of Software 2003 (FATES'03)*, number 2931 in Lecture Notes in Computer Science, pages 136–151. Springer–Verlag 2004.

Comments

I participated in the discussions, did the experiment measurements with my extension of UPPAAL, and wrote part of the paper.

Paper B:

Anders Hessel and Paul Pettersson. A Test Case Generation Algorithm for Real-Time Systems. In H-D. Ehrich and K-D. Schewe, editors, *Proceedings of the 9*th *International Conference on Quality Software 2004 (QSIC'04)*, pages 268–273, IEEE Computer Society Press, September 2004

Comments

I implemented the extension of UPPAAL described in the paper, did the experiments, took part in the discussions, and wrote part of the paper.

Paper C:

Johan Blom, Anders Hessel, Bengt Jonsson, and Paul Pettersson Specifying and Generating Test Cases Using Observer Automata In J. Gabowski and B. Nielsen, editors, *Proceedings of the 4th International Workshop on Formal Approaches to Testing of Software 2004 (FATES'04)*, volume 3395 in Lecture Notes in Computer Science, pages 125–139, Springer–Verlag Berlin Heidelberg 2005.

Comments

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

Other Work

Apart from the papers listed above, I have also participated in the following work:

- Anders Hessel, Kim G. Larsen, Brian Nielsen, Paul Pettersson, and Arne Skou. Time-Optimal Test Cases for Real-Time Systems. Invited presentation. In *Proceedings of the 1*st *International Workshop on Formal Modeling and Analysis of Timed Systems 2003 (FORMATS'03).*

- Johan Blom, Anders Hessel, Bengt Jonsson, Paul Pettersson. Specifying Test Cases Using Observer Automata. Extended abstract. In P. Pettersson and Wang Yi, editors, *Proceedings of the 16*th *Nordic Workshop on Programming Theory Oct 2004.* Technical Report 2004-041, ISSN 1404-3203. Department of Information Technology, Uppsala University, October 2004.

- Anders Hessel, Kim G. Larsen, Brian Nielsen, Paul Pettersson, and Arne Skou. Time-optimal Real-Time Test Case Generation using UPPAAL. Extended abstract, Real-Time in Sweden 2003 - RTiS'03 Västerås.

- Anders Hessel, Kim G. Larsen, Marius Mikucionis, Brian Nielsen, Paul Pettersson, and Arne Skou. Automated Model-Based Conformance Testing of Real-Time Systems. Book chapter. In J. Bowen, M. Harman, and R. Hierons, editors, *Formal Methods and Testing*, Springer Verlag. Submitted.

Contents

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 are 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.

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 overfill. 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- an 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

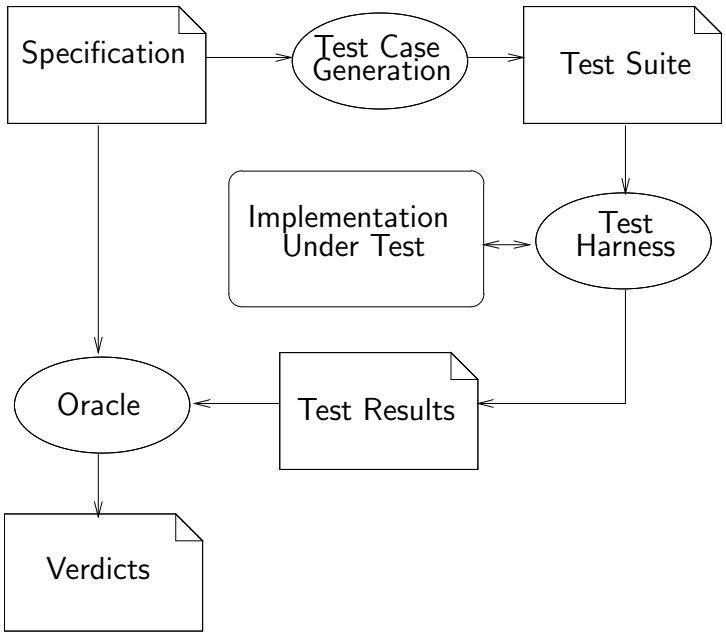


Figure 1: Black box testing

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

In Figure 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 of 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]. 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, the test suite would be affected by the specific implementation if white box testing is used.

Model-based testing is a black box technique where a model is used as specification. Often the model is examined by a tool to generate test suites. 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. Such a measure can be evaluated even before a test case is executed. As test case generation can be automated in model-based testing, the test cases do not have to be constructed by hand from the specification.

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 model. 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 model. 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 fulfills.

In model-based testing, 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 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.

In this thesis we will use test cases with only one expected continuation. A test harness that gives the verdict itself can abort the test case if an output of a system is not expected. Only if the full test case is executed the verdict pass is given. Our test cases will thus be a sequence of inputs and outputs, not necessarily alternating. In the case of timed systems, a delay will be specified between each input or output.

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. 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 completely abstracted away.

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 test cases are missed. If the model is too concrete, the construction of the model is as error prone as a full implementation. When model-based testing is used, the control states are typically in focus, and data are abstracted as much as possible. This is possible only if the data does not affect the control behavior of the implementation, 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. 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 "interger-even" with, e.g., 0 and 1. Only when all abstact 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, then 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 of all model-based testing.

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 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*

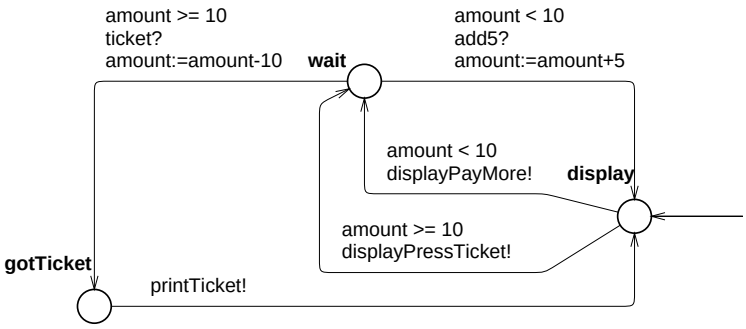


Figure 2: EFSM model of a parking ticket machine.

and get an output *printTicket* in return. Note that the *wait* location is used in three different states $\langle wait, 0\rangle$, $\langle wait, 5\rangle$, and $\langle wait, 10\rangle$. 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 $\sigma \in \mathbb{R}^X_{\geq 0}$ 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.

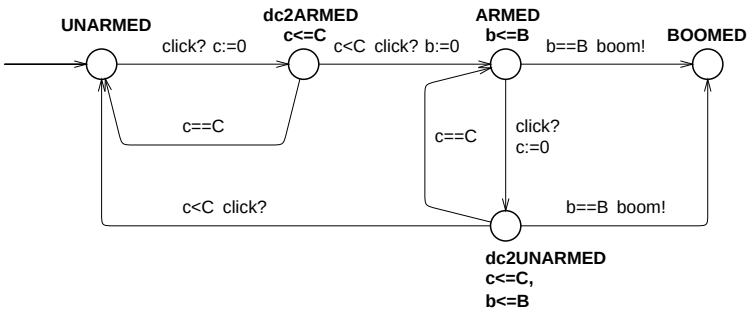


Figure 3: Timed automaton describing the function of the explosive pen in the movie Golden Eye

In Figure 3 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 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 C of this thesis, to explain the coverage criteria. Any 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.

Coverage criteria have so far been discussed in the context of coverage on code, but it is similar to measure coverage for a model. 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)

$$\texttt{if } (x == 2 \wedge y \geq 6)\ s1; \texttt{else } s2;$$

the decision is "$x == 2 \wedge y \geq 6$". It decides which of the branches of statements, $s1$ or $s2$, 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 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 $\{c_1 = true, c_2 = false\}$ and $\{c_1 = false, c_2 = false\}$, 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 $\{c_1 = true, c_2 = false\}$ and $\{c_1 = false, c_2 = true\}$. 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 the decision d_i both true and false without changing the other conditions in d_i. We will use the intermediate information $\langle c_j, k_j, true\rangle$ and $\langle c_j, k_j, false\rangle$, 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 $\langle c_j, k_j, true\rangle$ cannot exist without $\langle c_j, k_j, false\rangle$. 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 $\{c_1 = true, c_2 = false\}$ and $\{c_1 = false, c_2 = true\}$. All the possible CC coverage items are covered, i.e., $\langle c_1, true\rangle$, $\langle c_1, false\rangle$, $\langle c_2, true\rangle$, and $\langle c_2, false\rangle$. If MC/DC is applied the for the same test suite cover no items. If we add a case where $\{c_1 = false, c_2 = false\}$, then MC/DC will be covered, i.e., $\{c_1 = true, c_2 = false\}$, $\{c_1 = false, c_2 = false\}$ will give $\langle c_1\rangle$, because we have $\langle c_1, k_1, true\rangle$ and $\langle c_1, k_1, false\rangle$ where $k_1 = (false)$, only c_1 change. The second coverage item $\langle c_2\rangle$ (only c_2 change) is covered by $\{c_1 = false, c_2 = true\}$, $\{c_1 = false, c_2 = false\}$. 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 an important part of the standard coverage 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 discussion, 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, ..., e_n, e_u, where the variable x is not redefined in the sub-path e_1, ..., e_n, then the definition at e_d *reaches* the edge e_u. In this case the sub-path e_1, ..., 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 a statement $x := y + z$ the context is (e_y, e_z) 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 if k_i depends on the used variables in e_i. For used variables $v_1, \ldots, v_n$ in e_i, k_i consists of $e_1, \ldots, e_n$ 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 (e_1, e_2) where y was defined at e_2, and z at e_1, and the definition of y is 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 the test suite. This is not feasible for general EFSMs that can have arbitrary long paths or for code that has infinite loops. 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 anything more. The criterion requires that a test suite includes paths so that every definition has at least one use. 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. Notice that the number of uses do not affect the number of coverage items for the all-def criterion.

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 (x_1, e_1, e_2), 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 (x_2, e_2, e_3), 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 $\langle e_1, x_1, \ldots, x_{k-1}, e_k \rangle$ 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.

3.3 Coverage on Projected States

We have earlier described location coverage, i.e., to visit all locations in the model. In an EFSM, locations together with valuations of variables are used to define states. 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 4 a projected state machine graph is shown. We have projected the parking ticket machine from Figure 2 on the variable *amount*. The equivalence classes are $amount = 10$ and $amount \neq 10$. Notice that the

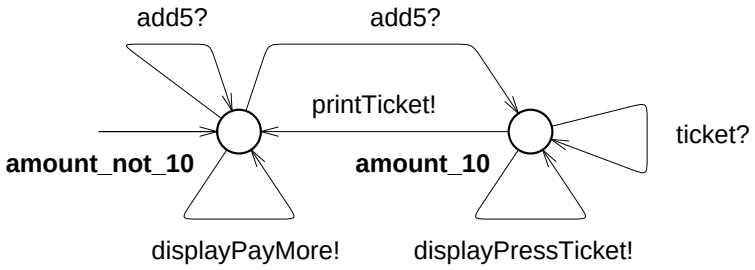


Figure 4: A projected state machine graph. The ticket machine in Figure 2 has been projected on the amount variable.

projection is non-deterministic, the *add*5 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 *add*5? to be covered, whereas there is only one edge with *add*5? in the original model. A test suite must cover both to add a five credit coin when the total amount will not sum up to ten credits and when it does sum up to ten.

If we project the parking ticket machine from Figure 2 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 where more variables used in the ticket machine the projection might have been useful to concentrate on the behavior around the *amount* variable.

4 Test Case Generation by Model Checking

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.

In the following, we will see how to transform the problem of test case

generation into a reachability problem. We will use witness traces to generate a collection of test cases that fulfills the requested coverage criterion. Each coverage item can be seen as a specific subproblem that can be solved independently. We will first describe how to generate a test case for one coverage item, and then show how make a test suite to cover a full coverage criterion. The main idea is to use a reachability property to determine if a coverage item can be fulfilled. We must thus construct a property that becomes true when a coverage item is fulfilled.

4.1 Adding Coverage Information to States

To evaluate a coverage item, we extend model states with auxiliary information. The purpose of the information is to determine whether a coverage item is fulfilled by the path to a state or not. We update the auxiliary information at each model transition. The actual coverage criterion determines the specific auxiliary information needed.

For the edge coverage criterion a coverage item for an edge e_i, denoted $\langle e_i \rangle$, is fulfilled if the edge has been traversed. In this case, when we traverse an edge e_i we update the auxiliary information with the coverage item $\langle e_i \rangle$. We let reachability properties include auxiliary information, and we can thus check the reachability of a state that includes a coverage item $\langle e_i \rangle$. If reachable, a witness trace for the coverage item can be generated.

Location and edge coverage are examples of coverage criteria that can be fulfilled momentarily, i.e., in a state or at a transition. For data-flow criteria this is not the case. We need to use information of the history of the path. Let $\langle x, e_d, e_u \rangle$ be a coverage item for a definition-use pair where x is a variable that is defined at edge e_d and used at edge e_u. To conclude that $\langle x, e_d, e_u \rangle$ is fulfilled when we traverse e_u we must know that the most recent definition of x was at e_d. Let $\langle x@e_1 \rangle$ represent that x is defined at e_1. This is added to the auxiliary information, but is replaced when x is defined at another edge.

We have now seen how to find a witness trace for a single coverage item. Note that a coverage item that once was fulfilled in a state will also be fulfilled in its successor states, thus the covered items increase monotonically. If we can find a state where all the items of a coverage criterion are covered, we can generate a test case that satisfy the full coverage criterion.

In general, it may not be possible to find a single path that fulfills all coverage items for a criterion. In this case we can combine different paths so that they together achieve the requested coverage. This can be done by adding a special "reset" transition to the model. The reset transition puts the model in its initial state, but keeps the achieved coverage items in the auxiliary information. Other information is discarded, e.g., where a variable was last defined.

If a model checker finds a path p_1 with coverage C_1, and then, after a

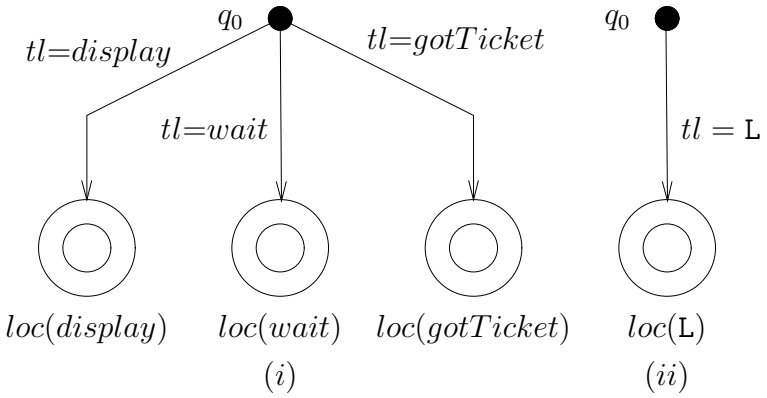


Figure 5: Two views on an observer for location coverage for the parking ticket machine in Figure 2

reset transition, finds a path p_2 with coverage C_2 the total coverage achieved will be $C_1 \cup C_2$ etc. If the coverage in $C_1 \cup C_2$ satisfies the property the model checker will produce a witness trace $p_1 \cdot reset \cdot p_2$. We assume that a reset transition can be recognized in the trace. We can now make a test suite consisting of a test case tc_1 based on p_1 and a test case tc_2 based on p_2.

4.2 Observers

There are many coverage criteria in the literature. A tool could implement support for them one by one, but this would not be very flexible. In Paper C of this thesis we suggest a language to formally specify coverage criteria using observer automata. The observer language is described for EFSM models, but other models such as timed automata would be applicable as well.

An observer is an automaton defined by locations and edges that are labeled with predicates. An observer automaton has an initial location marked is marked • and a set of *accepting* locations marked ◎. An observer collects, in a path, the auxiliary information that are used in the coverage criterion. The information must include status for all coverage items. Intuitively the observer follows the transition of an EFSM, and the set of possible observer locations is the auxiliary information.

An observer has always the possibility to stay in the initial location or in an accepting location. A location that is not an initial or an accepting location have to take an observer edge at each EFSM transition. If there are no enabled edges for a location, it is removed from the auxiliary information.

In Figure 5 two observers are shown for location coverage of the parking ticket machine in Figure 2. We recall that the parking ticket machine has three locations named *display*, *wait*, and *gotTicket*. The observer in

Figure 5 (i) has three accepting location, one for each possible EFSM location in Figure 2. After a transition in the EFSM, the predicate $tl{=}wait$ is satisfied if $wait$ was the target location of the EFSM transition. Observer location $loc(wait)$ represents coverage item $\langle wait \rangle$ for the location coverage criterion. It is similar for the other locations and the other edges.

In Figure 5 (ii) a parameterized observer is shown. If the parking ticket machine in Figure 2 is observed then the observer (i) and (ii) are equivalent. For Figure 5 (ii) the possible locations of the EFSM is the domain of parameter `L`, and thus there is one accepting observer location (and coverage item) for each location.

The parameterized observer has the advantage over the non-parameterized that the domains of the parameters do not have to be specified at the same time as the observer automaton. Thus a parameterized observer expresses a coverage criterion for any EFSM. A parameterized location represents the collection of locations obtained by instantiating its parameters, and similar for the edges.

Technically we superpose an observer onto an EFSM, so that each state are of the form $\langle \langle l, \sigma \rangle \parallel Q \rangle$, where $\langle l, \sigma \rangle$ is a state of the EFSM, and Q is a set of locations of the observer. A transition $\langle \langle l, \sigma \rangle \parallel Q \rangle \rightsquigarrow \langle \langle l', \sigma' \rangle \parallel Q' \rangle$ of this construct is possible if (i) a transition of the EFSM $\langle l, \sigma \rangle \rightsquigarrow \langle l', \sigma' \rangle$ is possible and (ii) Q' is a set of every possible q' such that there is an observer transition $q \rightsquigarrow q'$ where $q \in Q$. Observer transitions are done in response to an EFSM transition, thus the details are known about the EFSM transition (including the source and destination EFSM states). An observer location q in Q may have several or zero, possible transitions.

From the user's point of view an observer automaton can be seen as non-deterministic. None or many outgoing edges can be enabled from a parameterized location. The transition from Q to Q' is however deterministic. As the transition is calculated by subset construction, the underlying semantics are deterministic.

In Figure 6 an observer for the all-uses coverage criterion is shown. The initial location is marked q_0. The observer has one parameterized accepting location du. We use the convention to denote observer parameters with capital letters. The parameters can refer to variables, edges, locations, variable valuations, etc., in the model. In this example we use the `X` parameter for a variable, and the `E`, `E'` parameters for two edges. If a variable $amount$ is defined at edge e_1 in an EFSM transition, the observer edge with predicate $def(amount) \wedge edge = e_1$ will become true. It is then possible for the observer to make a transition from q_0 location to the $q_1(amount, e_1)$ location. As long as there is no redefinition of $amount$ in the EFSM, there is a possibility to stay in the location because of the self loop. If Q includes the location $q_1(amount, e_1)$ and $amount$ is used in an EFSM transition at edge e_2 a transition to the accepting location $du(amount, e_1, e_2)$ is possible, i.e., the coverage item $\langle amount, e_1, e_2 \rangle$ is fulfilled. Note that a transition from

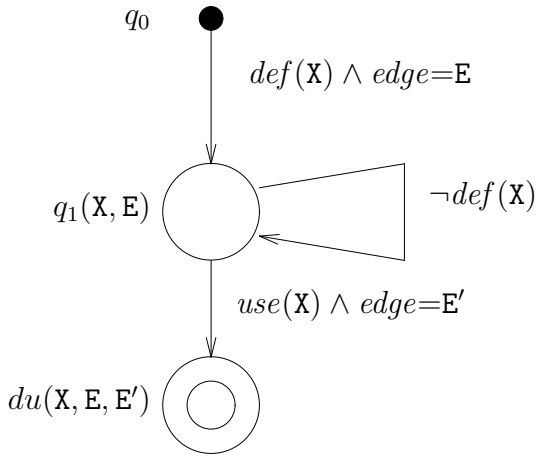


Figure 6: An all-uses (du-pair) observer

$q_1(amount, e_1)$ back to itself may also be possible by the self loop edge if *amount* is not defined in the EFSM transition, i.e., the definition of *amount* in e_1 may have other uses than in e_2 and the location $q_1(amount, e_1)$ should thus be kept in the auxiliary information.

4.3 Generating Test Suites from Timed Automata

We will assume that a specification is modeled as a network of timed automata compatible with the modeling language used in UPPAAL. Such models has no external actions, we therefore need to model both a system and its environment. This is usually done by making an unconstrained environment. Such environment can at any time synchronize with the external actions of the system model. In Figure 8 an environment is shown that has no restrictions on when it can synchronize with the *click* action except after a *boom* action has been received.

We will make a distinction between the specified system that we call *controller* and the environment of the systems that we call *environment*. Both the controller model and the environment model can consist of several automata, and internal communication is possible, both inside the controller and the environment model. The relation between the automata during model checking and real world testing is shown in Figure 7. 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 channel between the controller and environment in the modeling world has a corresponding communication medium in the real world.

To avoid generating test cases for which the system is not specified, the

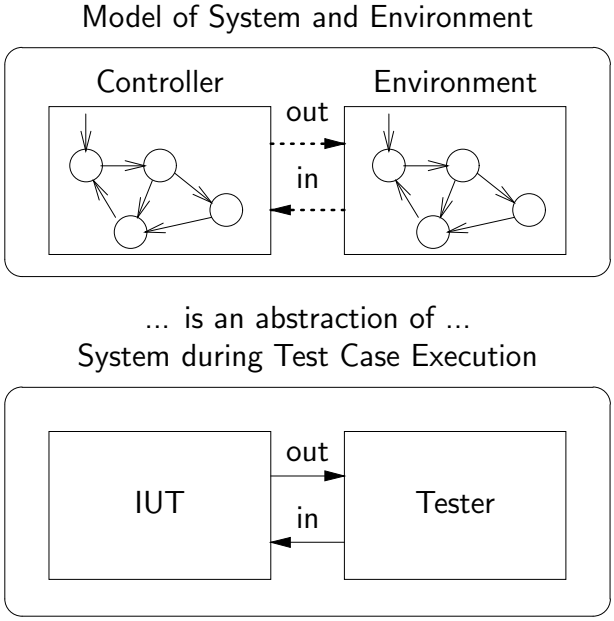


Figure 7: Model based testing

environment model can only provide the controller model with stimuli for which the system is specified. Stimuli outside the scope of the specification should not be provided. Other useful limitations to the environment models can be to model a tester with limitations, e.g., a tester that cannot provide stimuli infinitely fast.

A possible environment to generate test cases for the explosive pen in Figure 3 is shown in Figure 8. The environment can synchronize with the explosive pen on the *click*? action without timing constraints by the edge decorated with *click*!. When the explosive pen explodes the environment synchronize with the *boom* action. After the *boom* no more testing can be done and the system must be reset to continue.

Let $C = 2$ and $B = 10$, (seen from the side of the environment) the timed traces

TC 1: $click! \cdot 0 \cdot click! \cdot 10 \cdot boom?$

and

TC 2: $click! \cdot 2 \cdot click! \cdot 0 \cdot click! \cdot 0 \cdot click! \cdot 2 \cdot click! \cdot 0 \cdot click! \cdot 0 \cdot click! \cdot$
$0 \cdot click! \cdot 9 \cdot click! \cdot 1 \cdot boom?$

are two possible test cases. As all edges are traversed the all-edges coverage

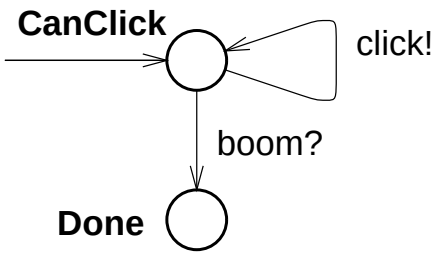


Figure 8: Timed automaton describing an possible environment to generate test cases for the explosive pen in Figure 3.

criterion is satisfied. The sequence of locations passed in the exercise for the test cases are for

TC 1: $UNARMED$, $dc2ARMED$, $ARMED$, $BOOMED$

and for

TC 2: $UNARMED$, $dc2ARMED$, $UNARMED$, $dc2ARMED$, $ARMED$, $dc2UNARMED$, $ARMED$, $dc2UNARMED$, $UNARMED$, $dc2ARMED$, $ARMED$, $dc2ARMED$, $BOOMED$.

When a witness trace is found for the required coverage, we convert the trace to a test case. The witness trace is a *timed trace*, i.e., an alternating sequence of inputs or outputs, and delays. UPPAAL provides us with a concrete trace, i.e., the trace has instantiated delays.

The trace can include transition that are internal for the controller or the environment. During test execution, we can only observe the interface of the implementation under test. A test case consists only of observable actions and delays. On the other hand the witness trace includes all transitions in the model. We project the trace on the interface between the controller and the environment, i.e., we remove all internal actions and transitions, and we sum up the delays between the remaining actions.

. If the remaining sequence has consecutive delays they are summed up to the delay between the observable actions.

When we provide inputs according to a timed trace generated by UPPAAL, we expect no deviation between the behavior of the system and the behavior in the timed trace. An unexpected response from the implementation under test could mean that the rest of the trace will be infeasible, because of the fact that the internal state of the system differs from the modeled state during generation. To be able to execute the generated test cases we restrict the timed automata model. Our restrictions are similar as the ones in [SVD01].

We call an automaton that fulfills our restrictions a *deterministic input enabled output urgent timed automata* (DIEOU-TA). For short the restrictions means that: (*i*) an input or a delay from one semantic state, leads only to one semantic state, (*ii*) no delay can be done when an input is offered, (*iii*) if the system can send an output no conflicting input, output, or delay is permitted.

To make a concrete test case we transform each action of the model in the trace to the real-world stimulus that the action models. The result is a concrete test case that can be executed by a test harness to test an implementation of the system.

We have not described what order to search the state space, i.e., which successor to generate at a given moment. The most common search orders used in reachability are *breadth first* and *depth first*. Besides search orders, Uppaal can be instructed to return the witness that has the *shortest trace* or the *fastest trace*. There are also support for heuristic algorithms that are variations of the A^* [BFH$^+$01, LBB$^+$01] algorithm. The A^* algorithms can speed up the generation performance if an admissible heuristics can be found.

By combining our encoding of coverage and the ability of Uppaal to find the fastest trace, we can generate test suites with a requested coverage that are time-optimal, i.e., no test suite with the same coverage can be executed faster. The generation is done by using the fastest trace algorithm, and saves time for the user that gets the implementation tested as fast as possible.

If we let time be a cost so that a time unit delay gives a cost of one cost unit, we can associate a constant cost with the reset transition. For paths including resets, we give the resets a cost for the "inconvenience". The reset cost could be very different depending on whether the whole system has to be restarted or if it is simple to reinitialize the system. The cost of a reset could be set very high, and thus the number of resets will be kept as low as possible for optimal test suite.

5 Summary of Papers

5.1 Paper A: Time-optimal Real-Time Test Case Generation using Uppaal

In this paper 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 the 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 an unnecessary

long time when executing the test cases. In this paper we also defines the deterministic input enabled output urgent timed automata.

We annotate the model to keep track of coverage items. The coverage is saved in model variables. If a state is found where all such variables are set, then the trace to that state is a trace which fulfills the coverage criteria. Thus, the problem of finding a trace that fulfills a coverage criteria has been reduced to a reachability problem. If it is impossible to find full coverage with one test case, we can decorate the model so that the system can be restored to its initial state, but keeping the coverage item information.

We also present some experiments on how this technique scales and on how the environment automata affect the time and memory used in the test case generation. A major drawback of this approach is that the model has to be annotated for every new coverage criterion. In Paper B we describe an implementations which makes such modifications superfluous.

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

In this paper we automate the test case generation of the coverage criteria that where described in Paper A. We present an abstract algorithm for symbolic reachability analysis with coverage and describes some aspects of an prototype implementation. We also show some experiments which demonstrate the benefits of a pruning technique we use.

In the paper we do not use any manually annotated auxiliary variables to store the information of the covered coverage items as in Paper A. Instead we keep track of the information automatically by extending the ordinary Uppaal state with additional data. We 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 cover items.

In the implementation, a bit-vector $\mathcal{B}$ is used to store the coverage items. If we already have explored an Uppaal state with coverage items $\mathcal{B}$ and find the same state again (or a subset of it as the clock evaluations are symbolic) but with coverage items $\mathcal{B}'$ we do not continue to search with the second state if its coverage items is a subset of the coverage items in the first. This is possible because we favor better coverage and more coverage will not destroy the property. In Paper A we stop the search only if we have equal coverage, i.e., $\mathcal{B} = \mathcal{B}'$.

The main criticism of Paper B is that only the criteria that are implemented in the tool are available to the user. It would be convenient if users could define their own coverage criteria. This is addressed in Paper C.

5.3 Paper C: Specifying and Generating Test Cases Using Observer Automata

In Paper C 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 as, e.g., timed automata. 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.

We assume that reset can be used as in Paper A to enable all coverage items to appear in one trace. Further we representat 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.

Even if observers has its most obvious benefits with data-flow coverage criteria they can also be used to express equivalence class-based coverage criteria as projection [FHNS02] of the current EFSM state. It is up to the implementer to supply macros and/or a language allowing users to create their own macros, e.g., for equivalence class partitions of state variable values. The observer language proposed in Paper C makes it possible to express coverage criteria that combines data-flow properties, e.g., a variable that is first defined and later used, and equivalence class projections, e.g., take the location from the state as parameter.

6 Related Work

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

6.1 Models

To make the automata deterministic Springintveld et al [SVD01] use similar restrictions as we do in Paper A. 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.

En-Nouaary et al. [ENDKE98] extends 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 an 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 often can find test cases without even generate 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. In [CKL98], Castanet et al. 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 A that this approach is valid in our setting.

In [CO00] Cardell-Oliver 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.

In [Kho02] Khoumsi 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 in [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) [AFT94], which is a determinizable subset of timed automata. ERA is basically timed automata where the is one clock per action and a clock is reset on the corresponding action. ERA is used by Nielsen and Skou in [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 what clocks are reset when. We consider the symbolic state space to be too large (even without the partition) to be generated for test case selection. Instead we select the test suite according to the coverage criteria. On the other hand some states might have to be visited several times before a coverage item is fulfilled. Thus, our state space is sometimes even larger because of the auxiliary information added. We use the same witness trace function from UPPAAL as [NS03].

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.

Krichen and Tripakis [KT04] uses 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 in [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).

Another work using online testing is [LMN05] by Larsen et al. In [LMN05] the tool T-UPPAAL is presented (now renamed to UPPAAL TRON). As our tool, 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 Observers

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

There is also a relation to 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.

6.3 Tools

There is a wide variety of test tool both in academia and in industry apart from the already mentioned tools. The STG tool [CJRZ02] a symbolic test generation tool using the IOSTS (Input Output Symbolic Transition System) [RdBJ00]. Rusu et al. generates test cases from test purposes in [RdBJ00] this is similar to our test purposes generated in Paper A. The STG tool is related to TorX [dBRS$^+$00] 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 in [BL03] – a tool-set for animations and test generation from B, Z, and statechart specifications.

Other more loosely related tools are tools for model checking of code. The Bandera Tool Set in [HD01] model check 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] by NASA.

7 Conclusions and Future Work

In this thesis we have shown how the problem of test case generation from timed automata models can be transformed into a reachability problem suitable for a model checker. By annotating the states with auxiliary informa-

tion about coverage, we can use UPPAAL to generate time-optimal test suites with an on-the-fly method.

We have also described a tool that can be used for test case generation without annotations in the model. Information about coverage items are automatically collected, for several coverage criteria. An improved algorithm for reachability of properties including coverage has been described.

Further, we have proposed a generic language to specify coverage criteria. The language is based on observer automata. We also show a method for generating test suites from EFSM models, given a coverage criterion specified with coverage observers.

Since an observer is an automaton with control structure, it is possible to specify, e.g., data-flow coverage criteria. The observer parameters make it possible to combine variables, locations, etc., from the current state into a projection, that can be a part of a coverage item. In this sense we combine two useful techniques in model-based testing.

We believe that the use of coverage observers simplifies the specification of coverage criteria. The language can be used to specify and combine most of the popular coverage criteria, as well as new criteria specified by users.

Currently we are developing a tool, called CO✓ER, for test case generation. The tool is a completely new version of the tool described in Paper B, and includes a tailored coverage observer language for networks of timed automata. The tool has been applied in a case study of a WAP gateway together with the telecom company Ericsson AB.

As future work, we will elaborate with different search strategies to remedy the blow up of the state space that comes with the coverage data added to the states. The CO✓ER tool contains a separate module for coverage handling, which could be plugged into other tools than UPPAAL. To specify a standard API for such a coverage module would be an opening for other tools to utilize the work, e.g., SPIN and UPPAAL TRON. Until now we have not been able to use clocks in observers. This together with other types of coverage that are not addressed yet as boundary coverage, condition coverage, etc., will be topics for future work.

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.

[AD94] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[AFT94] R. Alur, L. Fix, and Henzinger TA. Event-clock automata: a determinizable class of timed automata. In *Proc.* 6^{th} *Int. Conf. on Computer Aided Verification*, volume 818 of *Lecture Notes in Computer Science*. Springer–Verlag, 1994.

[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.* 11^{th} *International Static Analysis Symposium (SAS 2004)*, volume 3148 of *Lecture Notes in Computer Science*, pages 2–18. Springer–Verlag, 2004.

[BFH+01] G. Behrmann, A. Fehnker, T. Hune, K. G. Larsen, P. Pettersson, and J. Romijn. Efficient Guiding Towards Cost-Optimality in UPPAAL. In T. Margaria and W. Yi, editors, *Tools and Algorithms for the Construction and Analysis of Systems :* 7^{th} *International Conference, (TACAS'01)*, number 2031 in Lecture Notes in Computer Science, pages 174–188. Springer–Verlag, 2001.

[BL03] F. Bouquet and B. Legeard. Reification of executable test scripts in formal specification-based test generation: The java card transaction mechanism case study. In *FME 2003*, volume 2805 of *Lecture Notes in Computer Science*, pages 778–795. Springer–Verlag, 2003.

[BR01] T. Ball and S.K. Rajamani. Automatically validating temporal safety properties of interfaces. In M.B. Dwyer, editor, *Proc.* 8^{th} *International SPIN Workshop on Model Checking Software (SPIN'01)*, volume 2057 of *Lecture Notes in Computer Science*, pages 103–122. Springer–Verlag, 2001.

[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. Jéron, 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 : 8^{th} 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, Lousiana, 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 Steven J. Zeil. A formal evaluation of data flow path selection criteria. *IEEE Trans. on Software Engineering*, SE-15(11):1318–1332, November 1989.

[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 13^{th} 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 *19^{th} 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.

[HCL+03] H.S. Hong, S.D. Cha, I. Lee, O. Sokolsky, and H. Ural. Data flow testing as model checking. In *ICSE'03: 25^{th} Int. Conf. on Software Enginering*, pages 232–242, May 2003.

[HD01] John Hatcliff and Matthew Dwyer. Using the bandera tool set to model-check properties of concurrent java software. volume 2280 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 online test oracles from temporal logic specifications. *Int. Journal on Software Tools for Technology Transfer*, 4(4):456–471, 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 : 8^{th} 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 12^{th} 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.

[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: 7^{th} International Symposium, FTRTFT 2002*, volume 2469 of *LNCS*. Springer–Verlag, September 2002.

[KT04] M. Krichen and S. Tripakis. Black-box conformance testing for real-time systems. In Laurent Mounier Susanne Graf, editor, *Proc. 11^{th} International SPIN Workshop on Model Checking Software (SPIN'04)*, volume 2989 of *Lecture Notes in Computer Science*, pages 182–197. Springer–Verlag, 2004.

[LBB+01] K. G. Larsen, G. Behrmann, E. Brinksma, A. Fehnker, T. Hune, P. Pettersson, and J Romijn. As cheap as possible: Efficient cost-optimal reachability for priced timed automat. In G. Berry, H. Comon, and A. Finkel, editors, *Proc. of CAV 2001*, volume 2102 of *Lecture Notes in Computer Science*, pages 493–505. Springer–Verlag, 2001.

[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. 4^{th} 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.

[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.

[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. Jéron. An approach to symbolic test generation. In *Int. Conf. on Integrating Formal Methods*, volume 1945 of *Lecture Notes in Computer Science*, pages 338–357. Springer–Verlag, 2000.

[RW85] S. Rapps and E. J. Weyuker. Selecting software test data using data flow information. *IEEE Trans. Softw. Eng.*, 11(4):367–375, 1985.

[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.

[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:* 2^{nd} *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), 2003.

[vHW03] W. von Hagen and K. Wall. *The Definitive Guide to GCC*. Apress, 2003.

Paper A

Anders Hessel, Kim G. Larsen, Brian Nielsen, Paul Pettersson, and Arne Skou. Time-optimal Real-Time Test Case Generation using UPPAAL. In Alexandre Petrenko and Andreas Ulrich, editors, *Proceedings of the 3*rd *International Workshop on Formal Approaches to Testing of Software 2003 (FATES'03)*, volume 2931 in Lecture Notes in Computer Science, pages 136–151, Springer–Verlag 2004.

Time-optimal Real-Time Test Case Generation using Uppaal

Anders Hessel‡, Kim G. Larsen†, Brian Nielsen†, Paul Pettersson‡, and Arne Skou†
† Department of Computer Science, Aalborg University,
Fredrik Bajersvej 7E, DK-9220 Aalborg, Denmark
Email: {kgl,bnielsen,ask}@cs.auc.dk . ‡ Department of Information Technology,
Uppsala University,
P.O. Box 337, SE-751 05 Uppsala Sweden
Email: {hessel,paupet}@it.uu.se .

Abstract Testing is the primary software validation technique used by industry today, but remains ad hoc, error prone, and very expensive. A promising improvement is to automatically generate test cases from formal models of the system under test.

We demonstrate how to automatically generate real-time conformance test cases from timed automata specifications. Specifically we demonstrate how to *efficiently generate* real-time test cases with *optimal* execution time i.e test cases that are the *fastest* possible to execute. Our technique allows time optimal test cases to be generated using manually formulated test purposes or generated automatically from various coverage criteria of the model.

1 Introduction

Testing is the execution of the system under test in a controlled environment following a prescribed procedure with the goal of measuring one or more quality characteristics of a product, such as functionality or performance. Testing is the primary software validation technique used by industry today. However, despite the importance and the many resources and man-hours invested by industry (about 30% to 50% of development effort), testing remains quite ad hoc and error prone.

We focus on conformance testing i.e., checking by means of execution whether the behavior of some black box implementation conforms to that of its specification, and moreover doing this within minimum time. A promising approach to improving the effectiveness of testing is to base test generation on an abstract formal model of the system under test (SUT) and use a test generation tool to (automatically or user guided) generate and execute test cases. Model based test generation has been under scientific study for some time, and practically applicable test tools are emerging [BFG+00, Pel02, TB99, HLSU02]. However, little is still known in the context of real-time systems.

An important principal problem in generating real-time test cases is to compute *when* to stimulate the system and expect response, and to compute the associated correct verdict. This usually requires (symbolic) analysis of the model which in turn may lead to the state explosion problem. Another problem is *how to select* a very limited set of test cases to be executed from the extreme large number (usually infinitely many) of potential ones.

This paper demonstrates how it is possible to generate *time-optimal* test cases and test suites, i.e. test cases and suites that are guaranteed to take the least possible time to execute. The required behavior is specified using a deterministic and output urgent class of UPPAAL style timed automata. The UPPAAL model checking tool implements a set of efficient data-structures and algorithms for symbolic reachability analysis of timed automata. We then use the fastest diagnostic trace facility of the UPPAAL tool to generate time optimal test sequences. Test cases can either be selected through manually formulated test purposes or automatically from three natural coverage criteria—such as transition or location coverage–of the timed automata model.

Time optimal test suites are interesting for several reasons. First, reducing the total execution time of a test suite allows more behavior to be tested in the (limited) time allocated to testing. Second, it is generally desirable that regression testing can be executed as quickly as possible to improve the turn around time between software revisions. Third, it is essential for product instance testing that a thorough test can be performed without testing becoming the bottleneck, i.e., the test suite can be applied to all products coming of an assembly line. Finally, in the context of testing of real-time systems, we hypothesize that the fastest test case that drives the SUT to some state, also has a high likelihood of detecting errors, because this is a stressful situation for the SUT to handle.

The rest of the paper is organized as follows: Section 2 discusses related work, and Section 3 introduces our framework for testing real-time systems based on a testable subclass of timed automata. Section 4 and 5 describe how to encode test purposes and test criteria, and report experimental results respectively. Section 6 concludes the paper.

2 Related Work

Relatively few proposals exist that deal explicitly and systematically with testing real-time properties [Kho02, HNTC99, CKL98, SVD01, ENDKE98, CO00, CL97, MMM95, NS03]. In [CO00, ENDKE98, SVD01] test sequences are generated from a timed automata (TA) by applying variations of finite state machine (FSM) checking sequence techniques (see eg. [LY96]) to a discretization of the state space. Experience shows that this approach suffers seriously from the state explosion problem and resulting large number of test

sequences. The work in [HNTC99] and [Kho02] also use checking sequences, but is based on different structures and state verification methods. Both assume determinism, but not output urgency. To distinguish sequences that can always be executed to completion independent on output timing and sequences that may be executed to completion, [HNTC99] defines may- and must-traceability of transition sequences in a TA. The unique IO sequence (UIOv) method is then applied to a FSM derived from the TA by simply removing the clock conditions on transitions. The sequences are then checked for their may- and must-traceability, and the procedure is re-iterated when necessary. This may result in many iterations and in incomplete test-suites. The work in [Kho02] assumes a further restricted TA model where all transitions with the same observable action resets the same set of clocks. The TA is first translated into a (larger) alternative automaton where clock constraints are represented as set-timer and expire-timer events. Based on this, the generalized Wp method is used to compute checking sequences.

In most FSM based approaches, tests are *selected* based on a fault-model identifying implementation faults that is desired to be (or can be) detected during testing. Little or no evidence is given to support that the real-time fault models correspond to faults that occur frequently in practice. Another problem is the required assumptions about the number of states in the SUT, which in general is difficult to estimate. The coverage approach guarantees that the test suite is derived systematically and that it provides a certain level of thoroughness, which is important in industrial practice. It is important to stress that this is a practically founded heuristic *test selection* technique. Similarly, when time optimal sequences are generated, this is also a level of test selection, where only the fastest to execute are selected. Our goal is *not* full fault coverage that will in principle guarantee that the SUT is correct if it passes all generated tests.

A different approach to test generation and selection is [CKL98] where a manually stated test purpose is used to define the desired sequences to be observed on the SUT. A synchronous product of the test purpose and TA model is first formed and used to extract a symbolic test sequence with timing constraints that reach a goal state of the test purpose. This symbolic trace can be interpreted at execution time to give a final verdict. This work does not address test suite optimization or time optimality, does not address test generation without an explicit test purpose, and does not appear to be implemented in a tool. [NS03] proposes a fully automatic method for generation of real-time test sequences from a subclass of TA called event-recording automata which restricts how clocks are reset. The technique is based on symbolic analysis and coverage of a coarse equivalence class partitioning of the state space.

Our work is based on existing efficient and well proven symbolic analysis techniques of TA, and unlike others addresses time optimal testing. Most other work on optimizing test suites, e.g [ADLU91, UUFSA99, HLSU02],

focus on minimizing the length of the test suite which is not directly linked to the execution time because some events take longer to produce or real-time constraints are ignored. Others have used (untimed) model-checking tools to produce test suites for various model coverage criteria e.g., [HLSU02].

The main contributions of the paper are 1) application of time and cost optimal reachability analysis algorithms to the context of *time-optimal test case generation*, 2) an *automatic* technique to generate time optimal covering test suites for three *important coverage criteria*, 3) through creative use of the diagnostic trace facility of UPPAAL, a *test generation tool* exists that is based on efficient and well-proven algorithms, and finally 4) we provide *experimental evidence* in that the proposed technique has practical merits.

3 Timed Automata and Testing

We will assume that both the system under test (SUT) and the environment in which it operates are modeled as TA.

3.1 Timed Automata

Let X be a set of non-negative real-valued variables called *clocks*, and $Act = \mathcal{I} \cup \mathcal{O} \cup \{\tau\}$ a set of input actions $\mathcal{I}$ and output-actions $\mathcal{O}$, (denoted $a?$ and $a!$), and the non-synchronizing action (denoted τ). Let $\mathcal{G}(X)$ denote the set of *guards* on clocks being conjunctions of simple constraints of the form $x \bowtie c$, and let $\mathcal{U}(X)$ denote the set of *updates* of clocks corresponding to sequences of statements of the form $x := c$, where $x \in X$, $c \in \mathbb{N}$, and $\bowtie \in \{\leq, <, =, \geq\}$[1]. A *timed automaton* (TA) over (Act, X) is a tuple (L, ℓ_0, I, E), where L is a set of locations, $\ell_0 \in L$ is an initial location, $I : L \to \mathcal{G}(X)$ assigns invariants to locations, and E is a set of edges such that $E \subseteq L \times \mathcal{G}(X) \times Act \times \mathcal{U}(X) \times L$. We write $\ell \xrightarrow{g,\alpha,u} \ell'$ iff $(\ell, g, \alpha, u, \ell') \in E$.

The semantics of a TA is defined in terms of a timed transition system over states of the form $p = (\ell, \sigma)$, where ℓ is a location and $\sigma \in \mathbb{R}^X_{\geq 0}$ is a clock valuation satisfying the invariant of ℓ. Intuitively, there are two kinds of transitions: delay transitions and discrete transitions. In delay transitions, $(\ell, \sigma) \xrightarrow{d} (\ell, \sigma + d)$, the values of all clocks of the automaton are incremented with the amount of the delay, d. Discrete transitions $(\ell, \sigma) \xrightarrow{\alpha} (\ell', \sigma')$ correspond to execution of edges $(\ell, g, \alpha, u, \ell')$ for which the guard g is satisfied by σ. The clock valuation σ' of the target state is obtained by modifying σ according to updates u. We write $p \xrightarrow{\gamma}$ as a short for $\exists p'.\ p \xrightarrow{\gamma} p', \gamma \in Act \cup \mathbb{R}_{\geq 0}$. A timed trace is a sequence of alternating time delays and actions in Act.

[1] To simplify the presentation in the rest of the paper, we restrict to guards with non-strict lower bounds on clocks.

A *network of TA* $\mathcal{A}_1 \parallel \cdots \parallel \mathcal{A}_n$ over (Act, X) is defined as the parallel composition of n TA over (Act, X). Semantically, a network again describes a timed transition system obtained from those of the components by requiring synchrony on delay transitions and requiring discrete transitions to synchronize on complementary actions (i.e. $a?$ is complementary to $a!$).

3.2 Uppaal and Time Optimal Reachability Analysis

UPPAAL is a verification tool for a TA based modeling language. Besides dense clocks, the tool supports both simple and complex data types like bounded integers and arrays as well as synchronization via shared variables and actions. The specification language supports safety, liveness, deadlock, and response properties.

To produce test sequences, we shall make use of UPPAAL's ability to generate diagnostic traces witnessing a submitted safety property. Currently UPPAAL supports three options for diagnostic trace generation: *some trace* leading to the goal state, the *shortest trace* with the minimum number of transitions, and *fastest trace* with the shortest accumulated time delay. The underlying algorithm used for finding time-optimal traces is a variation of the A*-algorithm [BFH+01, LBB+01]. Hence, to improve performance it is possible to supply a heuristic function estimating the remaining cost from any state to the goal state.

Throughout the paper we use UPPAAL syntax to illustrate TA, and the figures are direct exports from UPPAAL. Initial locations are marked using a double circle. Edges are by convention labeled by the triple: guard, action, and assignment in that order. The internal τ-action is indicated by an absent action-label. Committed locations are indicated by a location with an encircled "C". A committed location must be left immediately as the next transition taken by the system. Finally, bold-faced clock conditions placed under locations are location invariants.

3.3 Deterministic, Input Enabled and Output Urgent TA

To ensure time optimal testability, the following semantic restrictions turn out to be sufficient. Following similar restrictions as in [SVD01], we define the notion of deterministic, input enabled and output urgent TA, DIEOU-TA, by restricting the underlying timed transition system defined by the TA as follows:

1. *Determinism.* Two transitions with the same label leads to the same state, i.e., for every semantic state $p = (\ell, \sigma)$ and action $\gamma \in Act \cup \{\mathbb{R}_{\geq 0}\}$, whenever $p \to \gamma p'$ and $p \to \gamma p''$ then $p' = p''$.

2. *(Weak) input enabled.* At any time any input action is enabled, i.e., whenever $p \to d$ for some delay $d \in \mathbb{R}_{\geq 0}$ then $\forall a \in \mathcal{I}.\ p \xrightarrow{a}$.

3. *Isolated Outputs.* If an output (or τ) is enabled then no other input or output transition is enabled, i.e., $\forall \alpha \in \mathcal{O} \cup \{\tau\}$. $\forall \beta \in \mathcal{O} \cup \mathcal{I} \cup \{\tau\}$ whenever $p \rightarrow \alpha$ and $p \rightarrow \beta$ then $\alpha = \beta$.

4. *Output urgency.* When an output (or τ) is enabled, it will occur immediately, i.e., whenever $p \rightarrow \alpha, \alpha \in \mathcal{O} \cup \{\tau\}$ then $p \not\rightarrow d, d \in \mathbb{R}_{\geq 0}$.

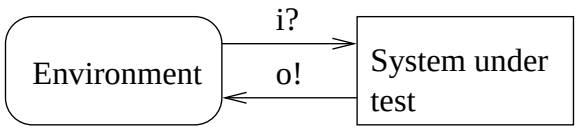


Figure 1: Test Specification

We assume that the test specification is given as a closed network of TA that can be partitioned into one subnetwork modeling the behavior of the SUT, and one modeling the behavior of its environment (ENV), see Figure 1. Often the SUT operates in specific environments, and it is only necessary to establish correctness under the (modeled) environment assumptions; otherwise the environment model can be replaced with a completely unconstrained one that allows all possible interaction sequences.

We assume that the tester can take the place of the environment and control the SUT via a distinguished set observable input and output actions. For the SUT to be testable the subnetwork modeling it should be *controllable* in the sense that it should be possible for an environment to drive the subnetwork model through all of its syntactical parts (e.g. transitions and locations). We therefore assume that the SUT specification is a DIEOU-TA, and that the SUT can be modeled by some unknown DIEOU-TA (this assumption is commonly refered to as the testing hypothesis). The environment model need not be a DIEOU-TA.

We use the simple light switch controller in Figure 2 to illustrate the concepts. The user interacts with the controller by touching a touch sensitive pad. The light has three intensity levels: `OFF`, `DIMMED`, and `BRIGHT`. Depending on the timing between successive touches (recorded by the clock x), the controller toggles the light levels. For example, in dimmed state, if a second touch is made quickly (before the switching time $T_{sw} = 4$ time units) after the touch that caused the controller to enter dimmed state (from either off or bright state), the controller increases the level to bright. Conversely, if the second touch happens after the switching time, the controller switches the light off. If the light controller has been in off state for a long time (longer than or equal to $T_{idle} = 20$), it should reactivate upon a touch by

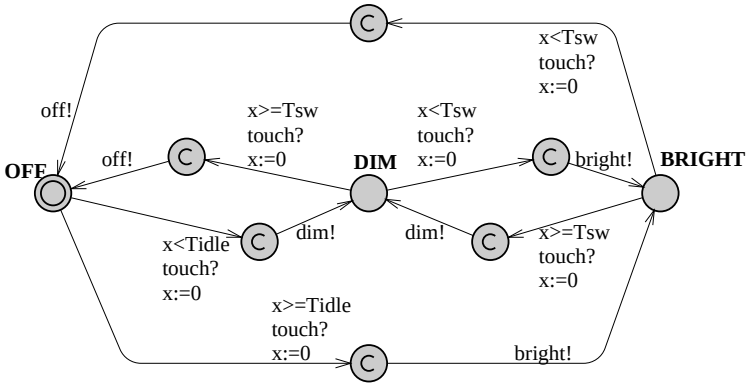


Figure 2: Light Controller

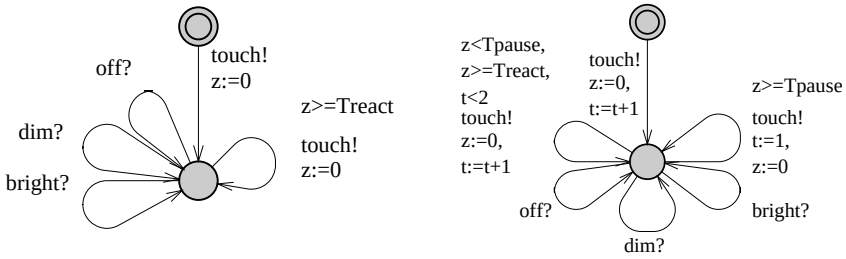


Figure 3: Two possible environment models for the simple light switch

going directly to bright level. We leave it to the reader to verify for herself that the conditions of DIEOU-TA are met by the model given.

The environment model shown in Figure 3(a) models a user capable of performing any sequence of touch actions. When the constant T_{react} is set to zero he is arbitrarily fast. A more realistic user is only capable of producing touches with a limited rate; this can be modeled setting T_{react} to a non-zero value. Figure 3(b) models a different user able to make two quick successive touches (counted by integer variable `t`), but which then is required to pause for some time (to avoid cramp), e.g., $T_{pause} = 5$.

3.4 From Diagnostic Traces to Test Cases

Let A be the TA network model of the SUT together with its intended environment ENV. A diagnostic trace produced by UPPAAL for a given reachability question on A demonstrates the sequence of moves to be made by

each of the system components and the required clock constraints needed to reach the targeted location. A (concrete) diagnostic trace will have the form:

$$(S_0, E_0) \rightarrow \gamma_0(S_1, E_1) \rightarrow \gamma_1(S_2, E_2) \rightarrow \gamma_2 \cdots (S_n, E_n)$$

where S_i, E_i are states of the SUT and ENV, respectively, and γ_i are either time-delays or synchronization (or internal) actions. The latter may be further partitioned into purely SUT or ENV transitions (hence invisible for the other part) or synchronizing transitions between the SUT and the ENV (hence observable for both parties).

For DIEOU-TA a *test sequence* is an alternating sequence of concrete delay actions and observable actions. From the diagnostic trace above a *test sequence*, λ, may be obtained simply by projecting the trace to the ENV-component, while removing invisible transitions, and summing adjacent delay actions. Finally, a *test case* to be executed on the real SUT implementation may be obtained from λ by the addition of *verdicts*.

Adding the verdicts require some comments on the chosen correctness relation between the specification and SUT. In this paper we require timed trace inclusion, i.e. that the timed traces of the implementation are included in the specification. Thus after any input sequence, the implementation is allowed to produce an output only if the specification is also able to produce that output. Similarly, the implementation may delay (thereby staying silent) only if the specification also may delay. The test sequences produced by our techniques are derived from diagnostic traces, and are thus guaranteed to be included in the specification.

To clarify the construction we may model the test case itself as a TA A_λ for the test sequence λ. Locations in A_λ are labeled using two distinguished labels, Pass and **fail**. The execution of a test case is now formalized as a parallel composition of the test case automaton A_λ and SUT A_S.

$$S \textbf{ passes } A_\lambda \;\; \textit{iff} \;\; A_\lambda \parallel A_S \not\rightsquigarrow \textbf{fail}$$

A_λ is constructed such that a *complete execution* terminates in a **fail** state if the SUT cannot perform λ and such that it terminates in a Pass state if the SUT can execute all actions of λ. The construction is illustrated in Figure 4.

4 Test Generation

4.1 Single Purpose Test Generation

A common approach to the generation of test cases is to first manually formulate a set of informal test purposes and then to formalize these such that the model can be used to generate one or more test cases for each test

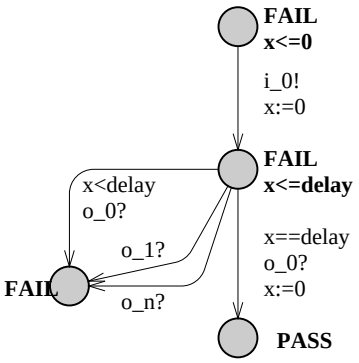


Figure 4: Test case automaton for the sequence $i_0! \cdot delay \cdot o_0?$

purpose. A test purpose is a specific test objective (or property) that the tester would like to observe on the SUT.

Because we use the diagnostic trace facility of a model-checker based on reachability analysis, the test purpose must be formulated as a property that can be checked by reachability analysis of the combined ENV and SUT model. We propose different techniques for this. Sometimes the test purpose can be directly transformed into a simple location reachability check. In other cases it may require decoration of the model with auxiliary flag variables. Another technique is to replace the environment model with a more restricted one that matches the behavior of the test purpose only.

TP1: Check that the light can become bright.

TP2: Check that the light switches off after three successive touches.

TP1 can be formulated as a simple reachability property:
`E<> LightController.bright` (i.e. eventually in some future the lightController automata enters location bright).

Generating the *shortest* diagnostic trace results in the test sequence: $20 \cdot touch! \cdot 0 \cdot bright?$. However, the *fastest sequence* satisfying the purpose is $0 \cdot touch! \cdot 0 \cdot dim? \cdot 0 \cdot touch! \cdot 0 \cdot bright?$.

TP2 can be formalized using the restricted environment model[2] in Figure 5 with the property `E<> tpEnv.goal`.

The fastest test sequence is $0 \cdot touch! \cdot 0 \cdot dim? \cdot 0 \cdot touch! \cdot 0 \cdot bright? \cdot 0 \cdot touch! \cdot 0 \cdot off?$.

[2]It is possible to use UPPAAL's committed location feature to compose the test purpose and environment model in a compositional way. Space limitations prevents us from elaborating on this approach.

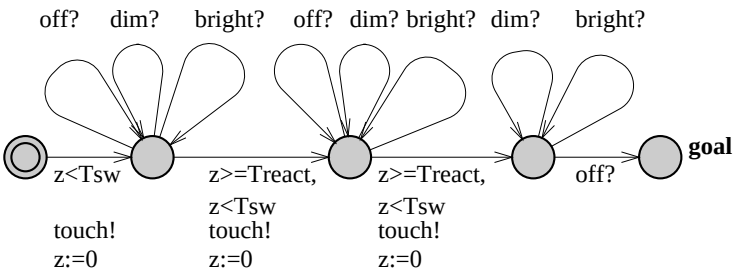


Figure 5: Test Environment for TP2

4.2 Coverage Based Test Generation

Often the tester is interested in creating a test suite that ensures that the specification or implementation is covered in a certain way. This ensures that a certain level of systemacy and thoroughness has been achieved in the test generation process. Here we explain how test sequences with guaranteed coverage of the SUT model can be computed using reachability analysis, effectively giving automated tool support. In the next subsection, we show how to generalize the technique to generate sets of test sequences.

A large suite of coverage criteria have been proposed in the literature, such as statement, transition, and definition-use coverage, each with its own merits and application domain. We explain how to apply some of these to TA models.

Edge Coverage: A test sequence satisfies the *edge-coverage criterion* if, when executed on the model, it traverses every edge of the selected TA-components. Edge coverage can be formulated as a reachability property in the following way: add an auxiliary variable e_i of type boolean (initially false) for each edge to be covered (typically realized as a bit array in UPPAAL), and add to the assignments of each edge i an assignment $e_i :=$ **true**; a test suite can be generated by formulating a reachability property requiring that all e_i variables are true: `E<>(` e_0`==true and` e_1 `==true` ... e_n`==true )`. The auxiliary variables are needed to enable formulation of the coverage criterion as a reachability property using the UPPAAL property specification language which is a restricted subset of CTL.

The light switch in Figure 2 requires a bit-array of 12 elements (one per edge). When the environment can touch arbitrarily fast the generated fastest edge covering test sequence has the accumulated execution time 28. The solution (there might be more traces with the same fastest execution time) generated by UPPAAL is:
EC: $0 \cdot touch! \cdot 0 \cdot dim? \cdot 0 \cdot touch! \cdot 0 \cdot bright? \cdot 0 \cdot touch! \cdot 0 \cdot off? \cdot 20 \cdot touch! \cdot$

$0 \cdot \mathit{bright}? \cdot 4 \cdot \mathit{touch}! \cdot 0 \cdot \mathit{dim}? \cdot 4 \cdot \mathit{touch}! \cdot 0 \cdot \mathit{off}?$.

Location Coverage: A test sequence satisfies the *location-coverage criterion* if, when executed on the model, it visits every location of the selected TA-components. To generate test sequences with location coverage, we introduce an auxiliary variable s_i of type boolean (initially false for all locations except the initial) for each location ℓ_i to be covered. For every edge with destination ℓ_i: $\ell' \xrightarrow{g,a,u} \ell_i$ add to the assignments u $s_i := \mathbf{true}$; the reachability property will then require all s_i variables to be true.

Definition-Use Pair Coverage: The definition-use pair criterion is a data-flow coverage technique where the idea is to cover paths in which a variable is *defined* (i.e. appears in the left-hand side of an assignment) and later is *used* (i.e. appears in a guard or the right-hand side of an assignment). Due to space-limitation, we restrict the presentation to clocks, which can be *used* in guards only.

We use (v, e_d, e_u) to denote a *definition-use pair* (DU-pair) for variable v if e_d is an edge where v is defined and e_u is an edge where v is used. A DU-pair (v, e_d, e_u) is valid if e_u is reachable from e_d and v is not redefined in the path from e_d to e_u. A test sequence covers (v, e_d, e_u) iff (at least) once in the sequence, there is a valid DU-pair (v, e_d, e_u). A test sequence satisfies the (all-uses) DU-pair coverage criterion of v if it covers all valid DU-pairs of v.

To generate test sequences with definition-use pair coverage, we assume that the edges of a model are enumerated, so that e_i is the number of edge i. We introduce an auxiliary data-variable v_d (initially **false**) with value domain $\{\mathbf{false}\} \cup \{1 \ldots |E|\}$ to keep track of the edge at which variable v was last defined, and a two-dimensional boolean array du of size $|E| \times |E|$ (initially **false**) to store the covered pairs. For each edge e_i at which v is defined we add $v_d := e_i$, and for each edge e_j at which v is used we add the conditional assignment $\mathit{if}(v_d \neq \mathbf{false})\mathit{then}\ du[v_d, e_j] := \mathbf{true}$. Note that if v is both used and defined on the same edge, the array assignment must be made before the assignment of v_d.

The reachability property will then require all $du[i, j]$ representing valid DU-pairs to be true for the (all-uses) DU-pair criterion. Note that a test sequence satisfying the DU-pair criterion for several variables can be generated using the same encoding, but extended with one auxiliary variable and array for each covered variable.

4.3 Test Suite Generation

Often a single covering test sequence cannot be obtained for a given test purpose or criterion (e.g. due to dead-ends in the model). To solve this problem, we allow for the model (and SUT) to be *reset* to its initial state, and to continue the test after the reset to cover the remaining parts. The generated test will then be interpreted as a test suite consisting of a set of

test sequences separated by resets (assumed to be implemented correctly in the SUT).

To introduce resets in the model, we shall allow the user to designate some locations as being reset-able. Obviously, performing a reset may take some time T_r that must be taken into consideration when generating time optimal test sequences. Reset-able locations can be encoded into the model by adding reset transitions leading back to the initial location. Let x_r be an additional clock used for reset purposes, and let ℓ be a reset-able location. Two reset-edges and a new location ℓ' must then be added from ℓ to the initial location ℓ_0, i.e.,

$$\ell \xrightarrow{reset!, x_r:=0} \ell'_{(x_r \leq T_r)} \xrightarrow{x_r == T_r, \tau, u_0} \ell_0$$

Here u_0 are the assignment needed to reset clocks and other variables in the model (excluding auxiliary variables encoding test purpose or coverage criteria [3]). If more than one component is present in either the SUT-model or environment model, the reset-action must be communicated atomically to all of them. This can be done using the committed location feature of UPPAAL. Further note that it may be possible to obtain faster (covering) test suites, if more reset-able locations are added, obviously depending on the time required to perform the reset, at the expense of increased model size.

4.4 Environment Behavior

A potential problem of the techniques presented above is that the generated test sequences may be non-realizable, in that they may require the environment of SUT to operate infinitely fast. In general, it is only necessary to establish correctness of SUT under the (modeled) environment assumptions. Therefore assumptions about the environment can be modeled explicitly and will then be taken into account during test sequence generation. In the following, we demonstrate how different environment assumptions affect the generated test sequences.

Consider an environment where the user takes at least 2 time units between each touch action; such an environment can be obtained by setting the constant T_{react} to 2 in Figure 3a. The fastest test sequences become:

TP1: $0 \cdot touch! \cdot 0 \cdot dim? \cdot 2 \cdot touch! \cdot 0 \cdot bright?$

TP2: $0 \cdot touch! \cdot 0 \cdot dim? \cdot 2 \cdot touch! \cdot 0 \cdot bright? \cdot 2 \cdot touch! \cdot 0 \cdot off?$.

Also reexamine the test suite **EC** generated by edge coverage, and compare with the one of execution time 32 generated when T_{react} equals 2:

EC': $0 \cdot touch! \cdot 0 \cdot dim? \cdot 4 \cdot touch! \cdot 0 \cdot off? \cdot 20 \cdot touch! \cdot 0 \cdot bright? \cdot 4 \cdot touch! \cdot 0 \cdot dim? \cdot 2 \cdot touch! \cdot 0 \cdot bright? \cdot 2 \cdot touch! \cdot 0 \cdot off?$.

[3] In the encoding of DU-pair coverage, the variables v_d should be set to **false** at resets.

When the environment is changed to the pausing user (can perform 2 successive quick touches after which he is required to pause for some time: reaction time 2, pausing time 5), the fastest sequence has execution time 33, and follows a completely different strategy, that ensures that one of the additional waiting times T_{pause} is overlapped with a position where the tester needed to wait anyway.

EC": $0 \cdot touch! \cdot 0 \cdot dim? \cdot 2 \cdot touch! \cdot 0 \cdot bright? \cdot 5 \cdot touch! \cdot 0 \cdot dim? \cdot 4 \cdot touch! \cdot 0 \cdot off? \cdot 20 \cdot touch! \cdot 0 \cdot bright? \cdot 2 \cdot touch! \cdot 0 \cdot off?$.

5 Experiments

In the previous section we presented techniques to compute time optimal covering test suites. In the following we show empirically that the performance of our technique is sufficient for practically relevant examples, and to indicate how heuristic search methods can be used to compute optimal or near optimal test cases from very large models. We are concerned with both the *execution time* of the generated test sequence, and the time and memory used to *generate* it.

5.1 The Touch Sensitive Switch

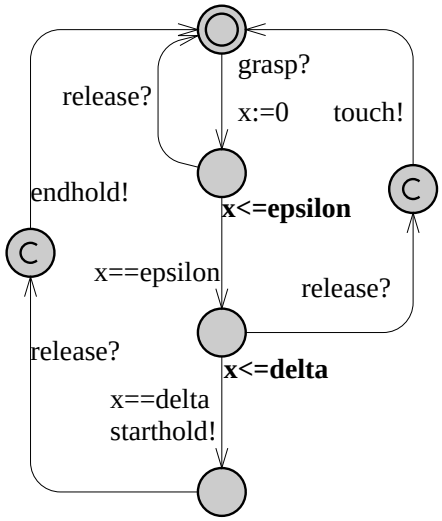


Figure 6: Interface Automaton

Most of the experiments reported here are based on a model of a touch

sensitive light switch (TSS). It has *Max* levels of brightness (0 corresponds to off). The lamp is operated by touching its wire, i.e. the wire can be grasped and released. The behavior of the controller can be expressed as follows: If the light is on, then a single grasp and release of the wire, will switch off the light. If the light is off, then a single grasp and release will switch on the light at the previous brightness level. Continuous holding of the wire increases the brightness (resp. decreases) if it was previously decreasing (resp. increasing). Once the maximum (resp. minimum) level is reached the brightness level decrease (resp. increase).

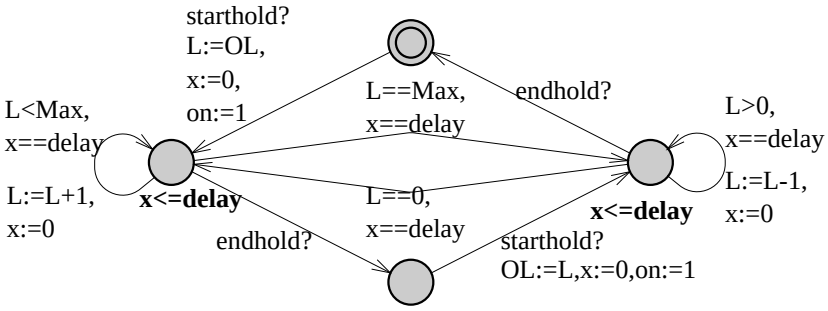


Figure 7: Dimmer Automaton

In reality a user can only perform two actions on the wire: `grasp` and `release`, and the time-separation between the two events is translated into either nothing (if the separation is very short), `touch` if it is short, and into a `starthold` and `endhold` pair if the separation is long. In the UPPAAL-model this translation is done by the interface component, shown in Figure 6. The dimmer component shown in Figure 7 reacts to `starthold` and `endhold` actions with a dimming effect. When changing the brightness level L, it is assumed that some maximum time ($Delay$) will elapse between two levels. The switch component shown in Figure 8(a) reacts to `touch` events by switching the light on to the previous light level OL, or off. The user is modeled in Figure 8(b).

We vary the model in two ways. First, the user may be *patient* or *impatient*. The impatient user insists on requiring interaction at least every $Wait = 15$ time units controlled by the invariant in user – this makes it harder for the user to change the intensity because he "gives up" the hold after just increasing the light one level. This invariant is removed in the patient user. Secondly, we vary the number of light levels from $Max = 10$ and up.

Table 9 shows the optimal execution times (in time units) for test suites

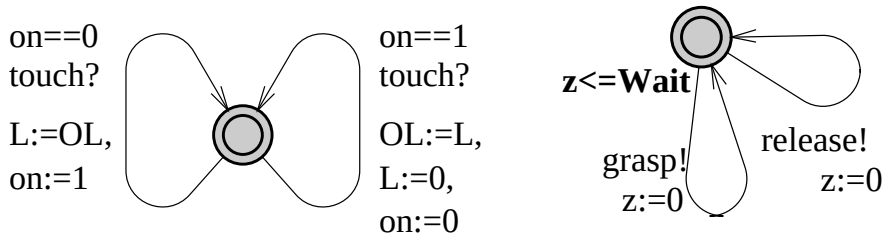


Figure 8: Switch (left) and User (right) Automata.

	Impatient		Patient	
Coverage Criterion	Exec-time	Suite length	Exec-time	Suite length
Location_{Dimmer}	20	12	20	12
$\text{Location}_{Dimmer,\ Switch,\ Interface}$	25	17	25	17
Edge_{Dimmer}	253	176	53	38
$\text{Edge}_{Interface}$	15	14	15	14
$\text{Edge}_{Dimmer,\ Switch,\ Interface}$	263	188	63	50
$\text{Edge}_{Interface}$+Location_{Dimmer}	25	19	25	19
Def-Use_{on}	40	34	40	34
Def-Use_{OL}	45	34	45	34

Table 9: Optimal execution time and suite length for various coverage criteria.

generated from different coverage criteria of the TSS, or selected subsets of components thereof, and the length (number of transitions) of the generated test suite. We notice that the patient user results in shorter and faster traces in our experiments.

5.2 System Size and Environment Behavior

To see how our technique scales, we increase the number of light levels in the TSS model. The result, listed in Table 10, shows that the particular example scales well: execution time (in time units), generation time, and memory usage for the impatient user increase essentially linearly with the number of light-levels. This is not surprising as the system size is varied by adjusting a counter, and not the number of parallel components.

It is more interesting to compare the patient and impatient user. Consider the system with 50 light levels. The optimal execution time for the impatient user is high (1183 time units), the reason being that the light

	Impatient			Patient		
Levels	Exec-time	Gen-time(s)	Mem (MB)	Exec-time	Gen-time (s)	Mem (MB)
10	263	2.06	9.1	63	3.19	10.1
20	493	3.68	11.4	93	12.40	20.1
30	723	5.29	12.6	123	28.17	40.4
50	1183	8.59	17.4	183	78.30	86.9
100	2333	16.76	28.0	333	339.52	314.9
200	4633	34.45	44.3	633	1494.35	1233.8
400	9233	66.03	77.1	N/A	>7000	>4180.6

Table 10: Cost of obtaining edge coverage of the TSS with increasing light levels.

level is increased only by one before he gives up, and starts the hold action again. Obtaining coverage therefore requires many interactions (trace of length 828). In contrast, the optimal execution time for the patient user 183 time units (and the trace length is 130). If we compare the generation time, it can be seen that it is much cheaper to compute the (very long) optimal solution for the impatient user than to compute the (short) optimal solution for the patient user.

Although this is surprising, there is a potential general explanation for this. The patient user environment poses no restrictions on the solution, and the test generator has complete freedom to find the optimal solution. This means that test generator has to evaluate all possible behaviors of this liberal environment. The impatient user is a more restricted environment, thus containing less possible behaviors. Therefore, searching the more liberal environment takes longer but also produces faster solutions.

There are two lessons to be learned. First, the relevance of an accurate model of the environment assumptions. Secondly, the use of the environment model to control test generation: restrict the environment to handle larger systems, but at the cost of more expensive solutions.

We have also created a DIEOU-TA version of the Philips audio control protocol [BPV94] frequently studied in the context of model checking. The system consists of a sender and a receiver communicating over a shared bus. The sender inputs a sequence of bits to be transmitted, Manchester encodes them, and transmits them as high and low voltage on the bus. Further, it checks for collisions by checking that the bus is indeed low when it is itself sending a low signal. The receiver is triggered by low-to-high transitions on the bus, and decodes the bits based on this information.

Table 11 summarizes the results. The first row contains results for the protocol tested with an environment consisting of a bus that may spontaneously go high to emulate a collision, and a sender buffer producing any legal input-bit sequence. The second row shows results for a receiver tested

Coverage Criterion	Execution time (μs)	Generation time (s)	Memory usage (KB)
Edge_{Sender}	212350	2.2	9416
$\text{Edge}_{Receiver}$	18981	1.2	4984
$\text{Edge}_{Sender,Bus,Receiver}$	114227	129.0	331408

Table 11: Results for the Philips audio protocol.

	First Sol.			Optimal Sol.	
Search order	Exec-time	Gen-time (s)	Memory (MB)	Gen-time	Mem (MB)
BF	123	27.91	40.8	N/A	N/A
DF	791	0.15	4.9	N/A	N/A
C_BF	123	30.44	42.6	31.31	43.3
C_DF	791	0.15	6.5	248.64	127.0
C_BF_R	123	30.70	42.6	30.87	42.9
C_DF_R	791	0.15	6.4	21.62	32.1
C_MC	123	25.87	39.3	26.19	39.5
C_MC_R	123	3.23	13.0	3.32	13.1

Table 12: Cost of edge coverage of TSS (Max=30) using different search orders.

in an environment consisting of a bus, and a buffer to hold the received bits. The third row is the results for the receiver tested in an environment consisting of a sender component with sender buffer, a bus, and receiver buffer. Thus the last row represents a rather large system. In all cases the time optimal covering test sequence could be computed in reasonable time.

5.3 Search-order and Guiding

UPPAAL allows the state space to be traversed in several different orders with different performance characteristics w.r.t. execution time of the generated test suite and the size of the system that can be handled. In particular, the A^* algorithm has potential significant impact. We here demonstrate how it can be employed for test generation to efficiently compute edge coverage in the TSS model.

The measured numbers are listed in Table 12. BF (DF) denotes breadth-first (depth-first) search order. The optimal execution time remains identical at 123 time units for all search orders. We note that using depth-first search during time optimal analysis (C_DF) UPPAAL produces (many) solutions quickly, but consumes long time to ultimately find the optimal one. During time optimal reachability analysis UPPAAL (symbolically) computes for each reached state the time C accumulated so far. Let C_g be the fastest time to a goal state found so far. When another state is found during exploration

with an accumulated time $C \geq C_g$ further exploration from that state is unnecessary, and the search can be pruned. Minimum accumulated time-first (MC) explores states ordered by their minimum accumulated time. To increase the efficiency further, it is possible to provide a safe estimate of the time that remains R from a given state to the goal state. Pruning can then be performed when a state is found with $C + R \geq C_g$. In Table 12 a search order combined with a remaining estimate is suffixed by an "_R".

It is easy to see in the dimmer component that the most time consuming edge to reach is the edge with guard $L = Max$. As estimate of remaining time, we use $(Max - L) \times delay$ if level $Max = L$ has not been reached, and 0 otherwise. Intuitively, the remaining time equals at least the number of light levels from Max value times the time to increase the light one level ($delay$). This formula has the feature that it can prune searches that turns back to lower light levels.

Compared to C_BF minimum accumulated time first search (C_MC) offers slightly improved generation time and memory usage. However, enabling remaining time estimate combined with this search order (C_MC_R) has a dramatic positive effect, and outperforms any of the other evaluated search orders.

6 Conclusions and Future Work

In this paper, we have presented a new technique for generating timed test sequences for a restricted class of timed automata. It is able to generate time optimal test sequences from either a single test purpose or a coverage criterion using the time optimal reachability feature of UPPAAL. Though a number of examples we have demonstrated how our technique works and performs. We conclude that it can generate practically relevant test sequences for practically relevant sized systems. However, we have also found a number of areas where our technique can be improved.

The DIEOU-TA model is quite restrictive, and a generalization will benefit many real-time systems. Especially, we are working on loosening the output urgency requirement. It may also be interesting to formulate coverage criteria that takes clock constraints into consideration.

Adding the required annotations for various coverage criteria by hand, and manually formulating the associated reachability property is tedious and error prone. We are working on a tool that performs these tasks automatically. Finally, we have found that the bit-vector annotations for tracking coverage and remaining time estimates may increase the state space significantly, and consequently also generation time and memory. The extra bits does not influence model behavior, and should therefore be treated differently in the verification engine. We are working on techniques that ignores these bits when possible, and that takes advantage of the coverage bits for

pruning states with “less” coverage.

References

[ADLU91] A. V. Aho, A. T. Dahbura, D. Lee, and M. Ü. Uyar. An Optimization Technique for Protocol Conformance Test Generation Based on UIO Sequences and Rural Chinese Postman Tours. *IEEE Transactions on Communications*, 39(11):1604–1615, 1991.

[BFG+00] M. Bozga, J-C. Fernandez, L. Ghirvu, C. Jard, T. Jéron, A. Kerbrat, P. Morel, and L. Mounier. Verification and Test Generation for the SSCOP Protocol. *Science of Computer Programming*, 36(1):27–52, 2000.

[BFH+01] G. Behrmann, A. Fehnker, T. Hune, K. G. Larsen, P. Pettersson, and J. Romijn. Efficient Guiding Towards Cost-Optimality in UPPAAL. In T. Margaria and W. Yi, editors, *Tools and Algorithms for the Construction and Analysis of Systems : 7^{th} International Conference, (TACAS'01)*, number 2031 in Lecture Notes in Computer Science, pages 174–188. Springer–Verlag, 2001.

[BPV94] D. Bosscher, I. Polak, and F. Vaandrager. Verification of an Audio-Control Protocol. In *Proc. of Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 863 of *Lecture Notes in Computer Science*, 1994.

[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, Lousiana, USA, October 12-15 1998. IEEE Computer Society Press.

[CL97] D. Clarke and I. Lee. Automatic Test Generation for the Analysis of a Real-Time System: Case Study. In *3rd IEEE Real-Time Technology and Applications Symposium*, 1997.

[CO00] R. Cardell-Oliver. Conformance Testing of Real-Time Systems with Timed Automata. *Formal Aspects of Computing*, 12(5):350–371, 2000.

[ENDKE98] A. En-Nouaary, R. Dssouli, F. Khendek, and A. Elqortobi. Timed Test Cases Generation Based on State Characterization Technique. In 19^{th} *IEEE Real-Time Systems Symposium (RTSS'98)*, pages 220–229, December 2–4 1998.

[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 : 8^{th} 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 12^{th} International Workshop on Testing Communicating Systems (IWTCS), September 1-3, 1999, Budapest, Hungary*, volume 147 of *IFIP Conference Proceedings*, pages 197–214. Kluwer, 1999.

[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: 7^{th} International Symposium, FTRTFT 2002*, volume 2469 of *LNCS*. Springer–Verlag, September 2002.

[LBB+01] K. G. Larsen, G. Behrmann, E. Brinksma, A. Fehnker, T. Hune, P. Pettersson, and J Romijn. As cheap as possible: Efficient cost-optimal reachability for priced timed automat. In G. Berry, H. Comon, and A. Finkel, editors, *Proc. of CAV 2001*, volume 2102 of *Lecture Notes in Computer Science*, pages 493–505. Springer–Verlag, 2001.

[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.

[NS03] Brian Nielsen and Arne Skou. Automated test generation from timed automata. *International Journal on Software Tools for Technology Transfer*, 5:59–77, 2003.

[Pel02] J. Peleska. Hardware/Software Integration Testing for the new Airbus Aircraft Families. In A. Wolis I. Schieferdecker, H. König, editor, *Testing of Communicating Systems XIV. Application to Internet Technologies and Services*, pages 335–351. Kluwer Academic Publishers, 2002.

[SVD01] J. Springintveld, F. Vaandrager, and P.R. D'Argenio. Testing Timed Automata. *Theoretical Computer Science*, 254(1-2):225–257, March 2001.

[TB99] J. Tretmans and A. Belinfante. Automatic testing with formal methods. In *EuroSTAR'99: 7^{th} European Int. Conference on Software Testing, Analysis & Review*, Barcelona, Spain, November 8–12, 1999. EuroStar Conferences, Galway, Ireland.

[UUFSA99] M. Ümit Uyar, M. A. Fecko, A. S. Sethi, and P. D. Amar. Testing Protocols Modeled as FSMs with Timing Parameters. *Computer Networks: The International Journal of Computer and Telecommunication Networking*, 31(18):1967–1998, 1999.

Paper B

Anders Hessel and Paul Pettersson. A Test Case Generation Algorithm for Real-Time Systems. In H-D. Ehrich and K-D. Schewe, editors, *Proceedings of the 9*th *International Conference on Quality Software 2004 (QSIC'04)*, pages 268–273, IEEE Computer Society Press, September 2004.

A Test Case Generation Algorithm for Real-Time Systems

Anders Hessel and Paul Pettersson
Department of Information Technology, Uppsala University,
P.O. Box 337, SE-751 05 Uppsala Sweden
Email: {hessel,paupet}@it.uu.se .

Abstract In this paper, we describe how the real-time verification tool Uppaal has been extended to support automatic generation of time-optimal test suites for conformance testing. Such test suites are derived from a network of timed automata specifying the expected behaviour of the system under test and its environment. To select test cases, we use coverage criteria specifying structural criteria to be fulfilled by the test suite. The result is optimal in the sense that the set of test cases in the test suite requires the shortest possible accumulated time to cover the given coverage criterion.

The main contributions of this paper are: (*i*) a modified reachability analysis algorithm in which the coverage of given criteria is calculated in an on-the-fly manner, (*ii*) a technique for efficiently manipulating the sets of covered elements that arise during the analysis, and (*iii*) an extension to the requirement specification language used in Uppaal, making it possible to express a variety of coverage criteria.

1 Introduction

In [HLN+04], we have presented a technique for generating time-optimal test suites from timed automata specifications using Uppaal [LPY97]. The technique describes how to annotate models with auxiliary variables so that test sequences from manually formulated test purposes or coverage criteria can be derived by reachability analysis. The result of the analysis is a diagnostic trace described as an alternating sequence of input actions and delays, which can be transformed into a (set of) test sequence(s) describing how to stimulate the system to fulfill the test criterion.

The tool presented in this paper, is a prototype version of the Uppaal tool based on the same technique but with the following extensions:

- a modified reachability analysis algorithm in which the coverage of a given criterion is collected during the reachability analysis performed by Uppaal, making manual model annotation superfluous.
- an implementation for efficiently representing the sets of covered elements that arises during the analysis. With the knowledge that such sets are always monotonically increasing along any trace of an automaton, it is safe to perform some pruning in the reachability analysis

normally not possible in model-checking (e.g. in case ordinary data-variables are used to annotate the model).

- a set of keywords representing coverage criteria extending the requirement specification language of UPPAAL.

The rest of this paper is organized as follow: in the next section, we describe the modeling language timed automata and the tool UPPAAL. In Section 3 we describe the algorithm implemented in the tool, in Section 4 we present the tool itself, and in Section 5 experiments are presented. Section 6 concludes the paper.

2 Preleminaries

We will use a restricted type of timed automata [AD94], extended with finite domain variables, called DIEOU-TA [HLN+04] to specify the system under test (SUT). The environment of the SUT is specified in the same way but without the DIEOU-TA restriction.

2.1 Timed Automata

A timed automaton is a finite state automaton extended with real-valued clocks. Let X be a set of non-negative real-valued *clocks*, and $Act = \mathcal{I} \cup \mathcal{O} \cup \{\tau\}$ a set of input actions $\mathcal{I}$ (denoted $a?$) and output-actions $\mathcal{O}$ (denoted $a!$), and a distinct non-synchronizing (internal) action τ. Let $\mathcal{G}(X)$ denote the set of *guards* on clocks being conjunctions of simple constraints of the form $x \bowtie c$, and let $\mathcal{U}(X)$ denote the set of *updates* of clocks corresponding to sequences of statements of the form $x := c$, where $x \in X$, $c \in \mathbb{N}$, and $\bowtie \in \{\leq, <, =, \geq\}$[1]. A *timed automaton* (TA) over (Act, X) is a tuple (L, ℓ_0, I, E), where L is a set of locations, $\ell_0 \in L$ is an initial location, $I : L \to \mathcal{G}(X)$ assigns invariants to locations, and E is a set of edges such that $E \subseteq L \times \mathcal{G}(X) \times Act \times \mathcal{U}(X) \times L$. We shall write $\ell \xrightarrow{g,\alpha,u} \ell'$ iff $(\ell, g, \alpha, u, \ell') \in E$.

The semantics of a TA is defined in terms of a timed transition system over states of the form $p = (\ell, \sigma)$, where ℓ is a location and $\sigma \in \mathbb{R}^X_{\geq 0}$ is a clock valuation satisfying the invariant of ℓ. Intuitively, there are two kinds of transitions: delay transitions and discrete transitions. In delay transitions, $(\ell, \sigma) \xrightarrow{d} (\ell, \sigma + d)$, the values of all clocks of the automaton are incremented with the amount of the delay, d. Discrete transitions $(\ell, \sigma) \xrightarrow{\alpha} (\ell', \sigma')$ correspond to execution of edges $(\ell, g, \alpha, u, \ell')$ for which the guard g is satisfied by σ. The clock valuation σ' of the target state is obtained by modifying σ according to updates u. We write $p \xrightarrow{\gamma}$ as a short for $\exists p'.\ p \xrightarrow{\gamma}$

[1] To simplify the presentation in the rest of the paper, we restrict to guards with non-strict lower bounds on clocks.

$p', \gamma \in Act \cup \mathbb{R}_{\geq 0}$. A timed trace is a sequence of alternating time delays and actions in Act.

A *network of TA* $\mathcal{A}_1 \parallel \cdots \parallel \mathcal{A}_n$ over (Act, X) is defined as the parallel composition of n TA over (Act, X). Semantically, a network again describes a timed transition system obtained from those of the components by requiring synchrony on delay transitions and requiring discrete transitions to synchronize on complementary actions (i.e. $a?$ is complementary to $a!$).

2.2 Deterministic, Input Enabled and Output Urgent TA

To ensure testability in the context of time, we require the following set of (sufficient) semantic restrictions on the SUT model. Following similar restrictions as in [SVD01], we define the notion of deterministic, input enabled and output urgent TA, DIEOU-TA [HLN+04], by restricting the underlying timed transition system defined by the TA as follows:

1. *Determinism.* For every semantic state $p = (\ell, \sigma)$ and action $\gamma \in Act \cup \{\mathbb{R}_{\geq 0}\}$, whenever $p \rightarrow \gamma p'$ and $p \rightarrow \gamma p''$ then $p' = p''$.

2. *(Weak) input enabled.* Whenever $p \rightarrow d$ for some delay $d \in \mathbb{R}_{\geq 0}$ then $\forall a \in \mathcal{I}.\ p \xrightarrow{a}$.

3. *Isolated Outputs.* $\forall \alpha \in \mathcal{O} \cup \{\tau\}.\ \forall \beta \in \mathcal{O} \cup \mathcal{I} \cup \{\tau\}$ whenever $p \rightarrow \alpha$ and $p \rightarrow \beta$ then $\alpha = \beta$.

4. *Output urgency.* Whenever $p \rightarrow \alpha, \alpha \in \mathcal{O} \cup \{\tau\}$ then $p \not\rightarrow d, d \in \mathbb{R}_{\geq 0}$.

2.3 UPPAAL and Testing

Uppaal [LPY97] is a tool for modeling and analysis of real-time systems[2]. Given a network of timed automata, extended with finite domain data variables, Uppaal can check if a given (symbolic) state is reachable from the initial state or not. If the state is reachable, the tool produces a diagnostic trace with action- and delay-transitions showing how the state can be reached.

It has been shown in [HLN+04] how to obtain a test sequence from a diagnostic trace of a DIEOU-TA. Given a network of timed automata consisting of a part modelling the system under test (SUT) and a part modeling the environment (ENV). The idea is to project the diagnostic trace to the visible actions between the SUT and the ENV part, and to sum up the delay transitions in between visible actions. The resulting test sequence can be converted to a test case which signals *fail* whenever the SUT does not behave according to the SUT model, i.e. produces unexpected output, or correct output at the wrong time-point.

[2] See the web site `http://www.uppaal.com/` for more details about the Uppaal tool.

```
PASS:= ∅
WAIT:= {((l_0, D_0), C_0)}
while WAIT≠ ∅ do
   select ((l, D), C) from WAIT
   if ((l, D), C) ⊨ φ_C then return "YES"
   if for all ((l, D'), C') in PASS: D ⊈ D' ∨ C ⊈ C' then
      add ((l, D), C) to PASS
      for all ((l_s, D_s), C_s)
         such that ((l, D), C) ⇝_c ((l_s, D_s), C_s):
         add ((l_s, D_s), C_s) to WAIT
return "NO"
```

Figure 1: An abstract algorithm for symbolic reachability analysis with coverage.

The technique presented in [HLN+04] shows how to transform a given test purposes or coverage criteria to annotations of the SUT and ENV models. For example, it shows the annotations and auxiliary variables needed so that definition-use pair coverage [FW88] be formulated as a reachability problem. The result is a diagnostic trace from which a set of test cases (a test suite) can be extracted which satisfies the definition-use pair coverage criteria in minimal time.

Whereas this is a viable technique, it is tedious and error prone in practice. The extra auxiliary variables also increase the size of the state space and thus the time and space required to perform the analysis. Since the extra variables do not influence the behaviour of the model, they should be treated differently. In the next section, we show how to move the auxiliary variables from the model into data structures in the analysis algorithm, and how they can be handled more efficiently.

3 Test Generation Algorithm

The reachability algorithm in UPPAAL is essentially a forward on-the-fly reachability algorithm that generates and explores the symbolic state space of a timed automata network. In the following we describe how the algorithm has been modified to check if a given coverage criteria is satisfied in a timed automata model.

3.1 Test Sequence Generation

The algorithm modified for generating test sequences is illustrated in Figure 4. The algorithm explores symbolic states of the form (l, D), where D is a zone (or DBM [Dil89]) representing a convex set of clock valuations, extended with a *coverage set* C representing the elements covered when the

state is reached. We use $(l, D) \rightsquigarrow (l', D')$ to denote a transition in the symbolic state space (see e.g. [BFH+01, LLPY97] for a description of the symbolic semantics implemented in UPPAAL). The algorithm terminates when the property φ_C is satisfied by a reached state. It is then possible to compute a diagnostic trace starting in the inital state and showing how to reach a state satisfying φ_C (see e.g. [LPY95]).

The algorithm in Figure 4 is similar to the ordinary reachability algorithm used in UPPAAL. The most significant modification is the addition of a coverage set C to the symbolic states. The particular representation of a coverage set depends on the coverage criteria mentioned in φ_C. The current implementation allows for conjunctions of *atomic coverage criteria* of the form $|A_l| \sim c$, $|A_e| \sim c$, or $|x_{du}| \sim c$, where $c \in \mathbb{N}$, $\sim \in \{>, \geq\}$, and $|A_l|$ and $|A_e|$ denotes the number of covered locations and edges in automaton A respectively, and $|x_{du}|$ the number of covered definition-use pairs of data variable x.

In the algorithm, the coverage sets are initiated to C_0 (line 2), checked for inclusion ("$\trianglelefteq$" on line 6), and then successors are generated (line 9). We define $((l, D), C) \rightsquigarrow_c ((l_s, D_s), C_s)$ iff $(l, D) \rightsquigarrow (l_s, D_s)$ and C is updated to C_s as follows:

- location coverage (in case φ_c contains an atomic coverage criterion of the form $|A_l| \sim c$): $C_s = C \cup \{ l_s\}$. In this case $C_0 = \{l_0\}$ and $C \trianglelefteq C'$ iff $C \subseteq C'$.

- edge coverage (in case φ_c contains an atomic coverage criterion of the form $|A_e| \sim c$): $C_s = C \cup \{e\}$, where $e \in E$ is the edge from which the transition $(l, D) \rightsquigarrow (l_s, D_s)$ is derived. In this case $C_0 = \{\}$ and $C \trianglelefteq C'$ iff $C \subseteq C'$.

- definition-use pair coverage on variable x (in case φ_c contains an atomic coverage criterion of the form $|x_{du}| \sim c$): In this case $C = \langle F, U\rangle$, where $F \in E \cup \{\bot\}$, and U is a coverage set of definition-use pairs of the form $\langle e_i, e_j\rangle$, where $e_i, e_j \in E$. We define $C_s = \langle F_s, U_s\rangle$:

$$F_s = \begin{cases} e & \text{if } x \text{ is defined on } e \\ F & \text{otherwise} \end{cases}$$

$$U_s = \begin{cases} U \cup \langle F, e\rangle & \text{if } F \neq \bot \text{ and } x \text{ is used on } e \\ U & \text{otherwise} \end{cases}$$

  where $e \in E$ is the edge from which the transition $(l, D) \rightsquigarrow (l_s, D_s)$ is derived. Initially $C_0 = \langle \bot, \{\}\rangle$ and $\langle F, U\rangle \trianglelefteq \langle F', U'\rangle$ iff $(F = F' \wedge U \subseteq U')$.

Thus, to check for location coverage the coverage set C is simply storing the set of locations that are visited when a symbolic state is reached. In a

network of timed automata, the update of C can easily be modified to check for coverage of a subset of the automata in the network. The case for edge coverage is similar. Definition-use pair coverage is checked by keeping track of active definitions in set F and covered DU-pairs in the set U.

Note how the coverage sets are checked for inclusion. Intuitively, the (symbolic) state $((l, D), C)$ does not need to be further examined if another state $((l, D'), C')$ is reached that contains all time-assignments, i.e. $D \subseteq D'$, and covers the same or more elements, i.e. $C \trianglelefteq C'$. This means that states with smaller coverage will not be further explored which is the reason for allowing only checks of lower bounds of the size of the coverage sets. The advantage is of course that the number of explored states becomes smaller, leading to faster termination of the algorithm (see Section 5 for more details).

To check $((l, D), C) \models \varphi_C$ in the algorithm is straight-forward. The value of $|A_l|$ or $|A_e|$ is simply the number of elements in C. For definition-use pair coverage, where C is a pair of the form $\langle F, U \rangle$ the value of $|x_{du}|$ is the number of elements in the set U.

3.2 Test Suite Generation

In [HLN+04] we describe a technique for generating test suites (set of test sequences) covering a given test criterion. The idea is to annotate the model with edges allowing the model to reset to its initial state (an updating the auxiliary variables accordingly). We now describe how the algorithm shown in Figure 4 (and Figure 2 below) can be modified in a similar way.

Assume a predicate $R \subseteq L \times \{0, 1\}$ which is true for all automaton locations that can be reset. In the algorithm, it suffices to insert the line

if $R(l)$ **then** add $((l_0, D_0), C)$ to WAIT

between line 8 and 9 of the algorithm of Figure 4. The case of definition-use pair coverage is the same except that $F = \{\}$ to indicate that no active definitions have yet been reached from the initial location.

It should be obvious that the effect of adding the line corresponds to allowing the system to reset to its initial symbolic state (l_0, D_0) but with the coverage collected before the reset. When UPPAAL returns a diagnostic trace the output is interpreted as a set of traces separated by the reset operations (which can be made visible in the diagnostic trace).

3.3 Time Optimal Test Suites

The standard reachability analysis algorithm implemented in UPPAAL has been extended to compute the trace with minimum time-delay satisfying a given reachability property [BFH+01]. In the same way as described above, the algorithm for time-optimal reachability can be extended to compute time-optimal test sequences. The resulting abstract algorithm is shown in

```
COST:= ∞
PASS:= ∅
WAIT:= {((l_0, D_0), C_0)}
while WAIT≠ ∅ do
    select ((l, D), C) from WAIT
    if ((l, D), C) ⊨ φ_C and min(D) < COST
        then COST := min(D)
    if for all ((l, D'), C') in PASS: D' ⋢ D ∨ C ⋢ C' then
        add ((l, D), C) to PASS
        for all ((l_s, D_s), C_s)
            such that ((l, D), C) ⇝_c ((l_s, D_s), C_s):
            add ((l_s, D_s), C_s) to WAIT
return COST
```

Figure 2: An abstract algorithm for symbolic time-optimal reachability analysis with coverage.

Figure 2. It should be noticed that the DBMs D used in the algorithm for time-optimal reachability is different from the one in ordinary symbolic reachability. For time-optimal reachability an extra clock is used that is never reset. The minimum value of this clock is the minimum time it takes to reach the state. We will not discuss this in detail here, but refer the reader to [BFH+01] for more details.

In [BFH+01], it is also shown how the algorithm can be extended with a set of techniques inspired by branch and bound algorithms [AC91]. Some of these extensions can also be applied when generating time-optimal test sequences, but it remains to be investigated in detail.

4 Implementation

In the algorithm(s) described in the previous section, the symbolic states contain a component representing the items covered in the path reaching the state. In the case of definition-use pair coverage, it also contains more information like the F set. In this section we will describe how the coverage sets have been implemented as bitvectors (in C++) in the algorithm.

We use bitvectors $\overline{v} \in \{0,1\}^n$ to implement a set C of n items, and associate a natural number i to each item to be covered. Then $v[i] = 1$ if item i is in the set. This is exactly the standard bitvector representation of sets. In order to improve efficiency, we use dynamic bitvectors and number the items as they are explored. For example, in the case of location coverage the locations are numbered in the order they are explored by the algorithm, and the length of the bitvector grows as new locations are explored.

4.1 Overview

An overview of the implemented coverage module is shown in Figure 3. The functionality of the module is to calculate (bitvector representation of) Q_s in a transition like $((\bar{l}, D), Q) \rightsquigarrow_c ((\bar{l}_s, D_s), Q_s)$. The input to the module is the coverage set C, a symbolic transition, and the new symbolic state, i.e. $\rightsquigarrow (\bar{l}_s, D_s)$. The example shown in the Figure 3 calculates Q_s for location coverage of an automaton P_1.

The module consists of three layers: the combined layer, the atomic layer, and the mapping table layer. The combined layer combines the coverage from the atomic terms and updates the set Q to Q_s. The atomic layer use the mapping tables to convert the coverage items found in the step to a bitvector. The layers in the architecture are fixed, but the configuration of the atomic layer differs, depending on the atomic coverage criteria used in the analysed property φ_c.

4.2 Layers

In general, the *AllCoverage* module consults one objects in the atomic layer for each atomic coverage term found in φ_c. In the illustrated example, there is only one atomic coverage — location coverage in automaton P_1. When an object in the atomic layer is consulted it is given a symbolic transition and a new state of the form $\rightsquigarrow (\bar{l}_s, D_s)$ and produces a bitvector δ_i with the bits set that correspond to items covered by the given transition and state. The successor set Q_s is created by *bitwise-or* of the old set Q and δ_i for all atomic objects. Thus, in the general case $Q_s = Q \cup (\bigcup_i \delta_i)$.

An object in the atomic layer is created for each atomic coverage described by the search property φ_c. In case of location or edge coverage it is instantiated with the corresponding type *location* or *edge*, and an automaton $\mathcal{A}$. In case of definition-use pair, the object is instantiated with the type *definition-use* and a variable name.

4.3 Dynamic Size of Bitvectors

The sets Q are saved with the symbolic state $(\bar{l}, D)$ as a bitvector that dynamically increases in size when new items are explored. Since long bitvectors are more expensive to manipulate we avoid to associate bits with items that have not yet been (or never will be) used. That is, the coverage items are numbered when they are first generated.

In the mapping table layer each coverage type has a table that associates the items with a unique bitnumber. To make the bitnumbers unique a global counter is used. The counter is incremented whenever a new item is found. In the example in Figure 3, the location is associated with the (bit)number 2 and thus the bitvector "10" is generated.

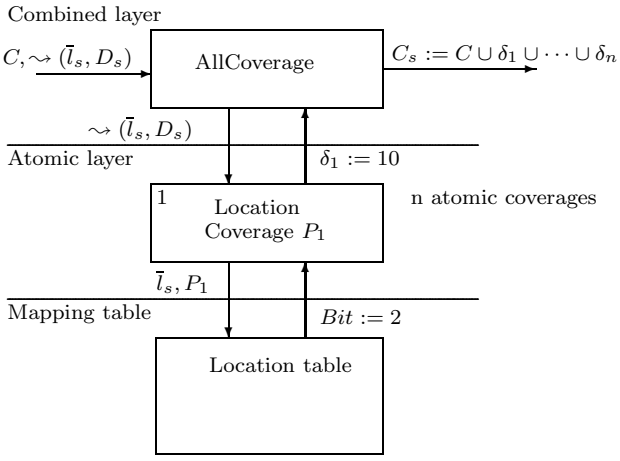


Figure 3: Architecture of the coverage module. Location coverage example instantiation.

5 Experiments

In order to evaluate the efficiency of the algorithm presented in this paper, we apply it to generate test suites from a model of protocol by Philips [BGK+96]. The protocol sends Manchester encoded bitstreams over a bus link, and detects collisions if two senders try to send that the same time-point. We use the model presented by Bengtsson et.al. in [BGK+96]. The model consists of seven timed automata. Four of the automata model the components of the protocol: two sender automata (*SA* and *SB*) , a receiver automaton *R*, and a wire automaton *wire*. Three of the automata model the environment of the protocol: two message automata providing the senders with messages (*messageA* and *messageB*), and an automaton checking the correctness of the received messages (*check*).

Table 4 shows the time and space required to generate time-optimal test suites with an coverage extended prototype version of UPPAAL implementing the algorithm described in Sections 3 and 4. The experiments have been performed on a Sun UltraSPARC-II 450MHz. Column "Pruned" gives the data when using the algorithm presented in this paper. Column "Original" gives the data when using bitvectors but not the extra pruning possible due to the monotonicity of the coverage sets (i.e. the effect of $\subseteq$). For both versions, we have used edge coverage criterion on two or three automata. We note that the reduction is 50 to 58% in time and 30 to 35% in memory consumption for this example.

	Original		Pruned	
Coverage criteria	Exec-time	Mem MB	Exec-time	Mem MB
$\|R_e\| \wedge \|SA_e\|$	8.91	18.5	4.43	13.0
$\|R_e\| \wedge \|SA_e\| \wedge \|SB_e\|$	14.61	25.5	6.06	16.5

Table 4: Measurements on Philips audio-control protocol with bus-collision.

6 Conclusion

In this paper, we have described how the real-time verification tool UPPAAL has been extended for test-case generation. In particular, we have extended the symbolic reachability analysis algorithm of the tool to generate traces satisfying simple coverage criteria, which can be used as test sequences or suites to test real-time systems.

The presented algorithm uses monotonically growing sets represented at bit-vectors to collect information about covered items. The monotonicity of these sets and the type of reachability properties checked for, allows for some pruning that normally can not be done. In our initial experiments, we have found the gained reduction in time and space consumed by the algorithm to be 50 to 58% and 30 to 35% respectively.

The current language for describing coverage criteria is very limited. As future work we will develop a more generic language which is not limited to a predefined set of criteria. Another possible future direction of work is to introduce monotonic variables in the modelling language of UPPAAL. Such variables might be useful in specifications of other problem areas such as e.g. scheduling and other planing problems.

References

[AC91] D. Applegate and W. Cook. A Computational Study of the Job-Shop Scheduling Problem. *OSRA Journal on Computing 3*, pages 149–156, 1991.

[AD94] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[BFH+01] G. Behrmann, A. Fehnker, T. Hune, K. G. Larsen, P. Pettersson, and J. Romijn. Efficient Guiding Towards Cost-Optimality in UPPAAL. In T. Margaria and W. Yi, editors, *Tools and Algorithms for the Construction and Analysis of Systems : 7^{th} International Conference, (TACAS'01)*, number 2031 in Lecture Notes in Computer Science, pages 174–188. Springer–Verlag, 2001.

[BGK+96] J. Bengtsson, W.O. David Griffioen, K. J. Kristoffersen, K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi. Verification of an Audio Protocol with Bus Collision Using UPPAAL. In R. Alur and T. A. Henzinger, editors, *Proc. 8^{th} Int. Conf. on Computer Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, pages 244–256. Springer–Verlag, 1996.

[Dil89] D. Dill. Timing Assumptions and Verification of Finite-State Concurrent Systems. In J. Sifakis, editor, *Proc. of Automatic Verification Methods for Finite State Systems*, volume 407 of *Lecture Notes in Computer Science*, pages 197–212. Springer–Verlag, 1989.

[FW88] P. G. Frankl and E. J. Weyuker. An applicable family of data flow testing criteria. *IEEE Trans. Softw. Eng.*, 14(10):1483–1498, 1988.

[HLN+04] A. Hessel, K. G. Larsen, B. Nielsen, P. Pettersson, and A. Skou. Time-Optimal Real-Time Test Case Generation using UPPAAL. In A. Petrenko and A. Ulrich, editors, *Proc. 3^{rd} International Workshop on Formal Approaches to Testing of Software 2003 (FATES'03)*, volume 2931 of *Lecture Notes in Computer Science*, pages 136–151. Springer–Verlag, 2004.

[LLPY97] F. Larsson, K. G. Larsen, P. Pettersson, and W. Yi. Efficient Verification of Real-Time Systems: Compact Data Structures and State-Space Reduction. In 18^{th} *IEEE Real-Time Systems Symposium (RTSS'97)*, pages 14–24. IEEE Computer Society Press, 1997.

[LPY95] K. G. Larsen, P. Pettersson, and W. Yi. Diagnostic Model-Checking for Real-Time Systems. In *Proc. of Workshop on Verification and Control of Hybrid Systems III*, volume 1066 of *Lecture Notes in Computer Science*, pages 575–586. Springer–Verlag, 1995.

[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.

[SVD01] J. Springintveld, F. Vaandrager, and P.R. D'Argenio. Testing Timed Automata. *Theoretical Computer Science*, 254(1-2):225–257, March 2001.

Paper C

Johan Blom, Anders Hessel, Bengt Jonsson, and Paul Pettersson Specifying and Generating Test Cases Using Observer Automata. In J. Gabowski and B. Nielsen, editors, *Proceedings of the 4th International Workshop on Formal Approaches to Testing of Software 2004 (FATES'04)*, volume 3395 in Lecture Notes in Computer Science, pages 125–139, Springer–Verlag Berlin Heidelberg 2005.

Specifying and Generating Test Cases Using Observer Automata

Johan Blom, Anders Hessel, Bengt Jonsson, and Paul Pettersson
Department of Information Technology, Uppsala University,
P.O. Box 337, SE-751 05 Uppsala Sweden Email:
{johan,hessel,bengt,paupet}@it.uu.se.

Abstract We present a technique for specifying coverage criteria and a method for generating test suites for systems whose behaviours can be described as extended finite state machines (EFSM). To specify coverage criteria we use observer automata with *parameters*, which monitor and accept traces that cover a given test criterion of an EFSM. The flexibility of the technique is demonstrated by specifying a number of well-known coverage criteria based on control- and data-flow information using observer automata with parameters. We also develop a method for generating test cases from coverage criteria specified as observers. It is based on transforming a given observer automata into a bitvector analysis problem that can be efficiently implemented as an extension to an existing state-space exploration such as, e.g. SPIN or Uppaal.

1 Introduction

Model based test case generation has in recent years been developed as a prominent technique in testing of reactive software systems. A model serves both the purpose of specifying how the system should respond to inputs from its environment, and of guiding the selection of test cases, e.g., using suitable coverage criteria. Typical notations for such models are state machines in some form, often extended with data variables. Test cases can be selected as individual "executions" of the model, checking that the outputs from the system under test (SUT) conform to those specified by the model.

There is a large literature and several tools (e.g., [dZ99, LBV02, SEG+98, MA00, dBRS+00]) for generation of test cases from extended state machine models (EFSMs). In typical approaches, the selection of test cases follows some particular coverage criterion, such as coverage of control states, edges, etc., or using an explicitly given set of test purposes [FJJV97, RdBJ00]. When the model contains data variables, constraint solving techniques can be used to find actual values of input parameters that drive the execution in a desired direction [LBV02, NS03, Meu00].

Since different coverage criteria are suitable in different situations, and satisfy different constraints on fault detection capability, cost, information about where potential faults may be located, etc., it is highly desirable that

a test generation tool is able to generate test suites in a flexible manner, for a wide variety of different coverage criteria. In other words, a test generation tool should accept a simple specification of a coverage criterion, given in a language that can easily specify a large set of coverage criteria, and be able to generate test suites accordingly.

In this paper, we present a technique for specifying coverage criteria in a simple and flexible manner, and a method for generating test cases according to such coverage criteria. The technique fits well as an extension of a state-space exploration tool, such as, e.g., SPIN [Hol97] or UPPAAL [LPY97], which performs enumerative or symbolic state-space exploration. It can also be used to generate monitors that measure the coverage of a specific test suite by monitoring the test execution.

In our technique, a coverage criterion is given as a set of *coverage items*, each of which represents an interesting structural property of the EFSM which should be examined by a test suite. A coverage item can state that a particular state, edge, or similar, should be visited, it can be an explicit test purpose, etc. Each coverage item is specified by an *observer*, which observes the execution of a test case, and reports acceptance when the test case has *covered* the coverage item that it specifies. For instance, a coverage item stating that a control state l of an EFSM model should be visited simply observes how the EFSM executes and reports acceptance when it enters l.

A typical coverage criterion is given as a (often rather large) set of coverage items. An important mechanism to facilitate specification of many coverage criteria is to allow *parameterization* of observers. In this way, one can specify a set of coverage items parameterized over, e.g., control states, data variables, edges, etc. of the EFSM model. Using this simple and general mechanism, we can specify most of the coverage criteria that have been used in the literature, and also tailor coverage to specific features of a particular SUT. For instance, if a particular interface is very error prone, we can specify a coverage criterion which requires all possible interleavings of interactions on that interface to be exhibited in a test suite.

A specification of a coverage criterion can be used for test suite generation using a state-space exploration tool. First, we superpose the coverage observers onto the EFSM, then we search for a test sequence or set of test sequences in which as many observers as possible report acceptance. For parameterized observers, we can record the achieved coverage by a (typically small) set of bitvectors, indexed by parameter values, which concisely represent the states of a large set of parameterized observers, in analogy with bitvector analysis in data-flow analysis, e.g., [Muc97]. The same machinery can also be used to monitor the achieved coverage of a certain test suite.

The remainder of the paper is structured as follows. We present EFSMs in the next section, and observers in Section 3. In Section 4, we show how our definitions of coverage can be used for test case generation, and report on a partial implementation of the technique. Section 5 concludes the paper.

Related Work. Most related work on test case generation from models of reactive systems employ some rather specific selection of coverage criteria. Explicitly given test purposes have been considered, both enumerative [FJJV97] and symbolic [RdBJ00]. Test purposes in these works can in some sense be regarded as coverage observers, but are not used to specify more generic coverage criteria and do not make us of parameterization, as in our work. For finite-state machines and EFSMs, several approaches focus on particular coverage criteria, e.g., Bouquet and Legeard [BL03] synthesize test cases corresponding to combinations of choices of control flow and boundary values of state variables, Nielsen and Skou [NS03] generate test cases that cover reachable symbolic states. These coverage criteria can be specified as observers in our framework.

Some approaches present more flexible techniques for specifying a variety of coverage criteria. Hong et al [HLSU02, HCL+03] describe how flow-based coverage criteria can be expressed in temporal logic. A particular coverage item is expressed in CTL, and a model checker generates a trace which covers the coverage item. In our approach, we use observers instead of temporal logic, which avoids some of the limitations of temporal logic [Wol81]. Friedman et al [FHNS02] specifies coverage by giving a set of projections of the state space (e.g., on individual state variables, components of control flow) that should be covered, possibly under some restrictions. Our approach generalizes this one, by allowing to define observers. Also, we can let one pass of a state-space exploration tool generate a test suite that covers a large set of coverage items, whereas the above approaches invoke a run of a model checker for each coverage item.

Constraint Logic Programming for model based test case generation has been used, e.g., by Marre and Arnould [MA00], by Meudec [Meu00], by Pretschner et al. [Pre01]. These approaches typically compile the specification into a constraint logic programming language, in which test cases can be extracted using symbolic execution.

2 Extended Finite State Machines

We assume that the specification of a module to be tested is given as an extended finite state machine in some syntax. In this section, we present a generic way to describe EFSMs, but our work can be adapted to more specific EFSM notations such as, e.g., UML Statecharts [GLM02] or SDL [ITU99].

We assume that a System Under Test (SUT) interacts with its environment through events. Whenever the SUT receives an input event, it responds by performing some local computation and emitting an output event. To a given SUT, we associate a set A of *event types*, each with a fixed *arity*. An *event* is a term of form $a(d_1, \ldots, d_k)$ where a is an event type of arity k

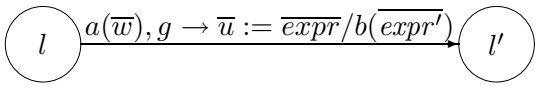
and $d_1, \ldots, d_k$ are the parameters of the event. The set A of event types is partitioned into *input event types* and *output event types*. A *trace* is a finite sequence

$$a_1(\overline{d}_1)/b_1(\overline{d}'_1) \quad a_2(\overline{d}_2)/b_2(\overline{d}'_2) \quad \cdots \quad a_n(\overline{d}_n)/b_n(\overline{d}'_n)$$

of input/output event pairs. Intuitively, the trace represents a behavior where the SUT, starting from its initial state, receives the input event, $a_1(\overline{d}_1)$ and responds with the output event $b_1(\overline{d}'_1)$. Thereafter, it receives the input event $a_2(\overline{d}_2)$ and so on. An *input sequence* is a finite sequence of input events.

Assume a set A_I of input event types, and a set A_O of output event types. An *Extended Finite State Machine* (EFSM) over (A_I, A_O) is a tuple $\langle L, l_0, \overline{v}, E \rangle$ where

- L is a finite set of *locations* (aka control states).
- $l_0 \in L$ is the *initial location*.
- $\overline{v}$ is a finite set of *state variables*.
- E is a finite set of *edges*, each of which is of form



  where

  - e is the name of the edge,
  - l is the source location, and l' is the target location,
  - $a \in A_I$ is an input event type, and $\overline{w}$ is a tuple of formal parameters of a,
  - g is a guard,
  - $\overline{u} := \overline{expr}$ is an assignment of new values to a subset $\overline{u} \subseteq \overline{v}$ of the state variables, and
  - $b(\overline{expr'})$ is an expression which evaluates to an output event.

  g, $\overline{expr}$, and $\overline{expr'}$ may depend on the formal parameters $\overline{w}$ of the input event and the state variables $\overline{v}$.

Intuitively, an edge of the above form denotes that whenever the EFSM is in location l and receives an event of form $a(\overline{w})$, then, provided that the guard g is satisfied, it can perform a computation step in which it updates its state variables by $\overline{u} := \overline{expr}$, emits the output event $b(\overline{expr'})$ and moves to location l'. We require the EFSM to be *deterministic*, i.e., that for any two edges with the same source location l and parameterized input event $a(\overline{w})$, the corresponding guards are inconsistent.

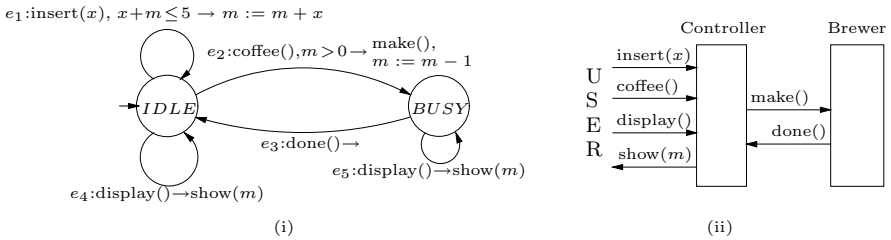


Figure 1: An EFSM specifying the controller of a simple coffee machine.

A *system state* is a tuple $\langle l, \sigma \rangle$ where l is a location, and σ is a mapping from $\overline{v}$ to values. We can extend σ to a partial mapping from expressions over $\overline{v}$ in the standard way. The *initial system state* is the tuple $\langle l_0, \sigma_0 \rangle$ where l_0 is the initial location, and σ_0 gives a default value to each state variable. A *computation step* is of the form $\langle l, \sigma \rangle \xrightarrow{a(\overline{d})/b(\overline{d}')} \langle l', \sigma' \rangle$ consisting of system states $\langle l, \sigma \rangle$ and $\langle l', \sigma' \rangle$, an input event $a(\overline{d})$, and an output event $b(\overline{d}')$, such that there is an edge of the (above) form $l \xrightarrow{a(\overline{w}), g \rightarrow \overline{u} := \overline{expr}/b(\overline{expr'})} l'$, for which $\sigma(g[\overline{d}/\overline{w}])$ is true, $\sigma' = \sigma[\overline{u} \mapsto \sigma(\overline{expr}[\overline{d}/\overline{w}])]$, and $\overline{d}' = \sigma(\overline{expr'}[\overline{d}/\overline{w}])$. A *run* of the EFSM over a trace $a_1(\overline{d}_1)/b_1(\overline{d}'_1) \cdots a_n(\overline{d}_n)/b_n(\overline{d}'_n)$ is a sequence of computation steps

$$\langle l_0, \sigma_0 \rangle \xrightarrow{a_1(\overline{d}_1)/b_1(\overline{d}'_1)} \langle l_1, \sigma_1 \rangle \xrightarrow{a_2(\overline{d}_2)/b_2(\overline{d}'_2)} \cdots \xrightarrow{a_n(\overline{d}_n)/b_n(\overline{d}'_n)} \langle l_n, \sigma_n \rangle$$

labelled by the input-output event pairs of the trace.

Example 1 *In Figure 1 an EFSM (from [HLSU02]) specifying the behavior of the controller of a simple coffee machine which interacts with a user and a brewer unit is shown. The controller has* $L = \{IDLE, BUSY\}$, $l_0 = IDLE$, $\overline{v} = \{m\}$, $A_I = \{insert, coffee, display, done\}$, $A_O = \{show, make\}$, *and* $E = \{e_1, e_2, e_3, e_4, e_5\}$. *The parameter* x *and the variable* m *take values that are integers in the range* $[0 \ldots 5]$.

An EFSM can be used to check that a trace of a SUT conforms to its specification, by checking that each output event produced by the SUT conforms to the corresponding output event prescribed in the EFSM. For test generation, the output events will not be significant, and we will therefore omit them in the rest of the paper, thus writing an edge of an EFSM as $l \xrightarrow{a(\overline{w}), g \rightarrow \overline{u} := \overline{expr}} l'$. We can also consider specifications that are parallel compositions of EFSMs, but omit such a treatment in this version of the paper.

3 Observers

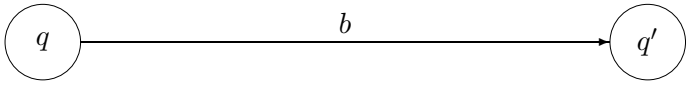
In this section, we present how to use observers to specify coverage criteria for test generation or test monitoring. A coverage criterion typically consists of a (long) list of items that should be "covered" or "visited". For instance, the criterion of "full location coverage" stipulates that a test suite should visit all locations of a given EFSM. We will use the term *coverage item* for an item that should be "covered" or "visited". Letting a test sequence be represented as a trace, we can use standard techniques from model-checking and run-time verification [VW86, HR02] to represent a coverage item by an *observer*, which monitors a trace and "accepts" whenever the coverage item has been covered. An observer observes how an EFSM executes a run over a trace, and "remembers" some chosen aspects of the EFSM execution. The observer can observe the events of the trace, as well as syntactical components of edges that the EFSM traverses in response to observed events, but should not interfere with the execution of the system.

Typical coverage criteria consist not only of a single coverage item, but of a large set of coverage items. We therefore extend the notion of observers by a parameterization mechanism so that they can specify a *set of* coverage items. Parameterized observers are simply observers, in which locations and edges may be parameterized by parameters that range over given domains. Each choice of parameter values gives a certain observer location or edge. For each specified coverage item, the observer has an *accepting* (possibly parameterized) location which (for convenience) we give the name of the corresponding coverage item. When the accepting location is entered, the trace has covered the corresponding coverage item.

As a very simple example, the coverage item *"visit location l of the EFSM"* can be represented by an observer with one initial state, and one accepting location, named $loc(l)$, which is entered when the EFSM enters location l. The coverage criterion *"visit all locations of the EFSM"* can be represented by a parameterized observer with one initial state, and one parameterized accepting location, named $loc(\texttt{L})$, where `L` is a parameter that ranges over locations in the EFSM. For each value l of `L`, the location $loc(l)$ is entered when the EFSM enters location l.

Formally, an *observer* is a tuple (Q, q_0, Q_f, B) where

- Q is a finite set of *observer locations*
- q_0 is the *initial observer location.*
- $Q_f \subseteq Q$ is a set of *accepting observer locations*, whose names are the corresponding coverage items.
- B is a set of edges, each of form



where b is a predicate that can depend on the input event received by the SUT, the mapping from state variables of EFSM to their values after performing the current computation step, and the edge in the EFSM that is executed in response to the current input event.

Intuitively, at any specific instant during test execution the observer is in one of its locations, q say. At each occurrence of an event, the observer traverses an outgoing edge from q, whose predicate is satisfied for this event, and the corresponding transition performed by the EFSM. Note that, in contrast to EFSMs, observers may be non-deterministic, since a coverage item in general can be covered in several ways.

In many cases, the initial location q_0 has an edge to itself with the predicate $true$. We use the symbol $\bullet$ to represent q_0 together with such a self-loop. Similarly, we assume that each $q_f \in Q_f$ has an edge to itself with the predicate $true$. We use the symbol $\circledcirc$ to represent accepting locations. In section 3.2, we discuss the effect of these self-loops in more detail. Intuitively, the one in q_0 is often used to allow the observer to non-deterministically start monitoring at any point of an EFSM run. The loop in each q_f is used to allow an observer to stay in an accepting location.

In order for observers to specify coverage criteria consisting of several coverage items, we allow locations and edges to be parameterized. Each parameter has a finite domain, which could be the set of EFSM locations, edges, state variables, or similar. We use uppercase letters in typewriter font for parameters. A parameterized location represents the collection of locations obtained by instantiating its parameters, and similarly for edges.

3.1 Observer Predicates

In the following we introduce a more specific syntax for the predicates b occurring on observer edges. The predicates will use a set of predefined *match variables* that are given values at the occurrence of

- an event $a(\overline{d})$,
- an edge $e : l \xrightarrow{a(\overline{w}),g \to \overline{u} := \overline{expr}} l'$ of the EFSM, traversed in response to $a(\overline{d})$,
- the computation step $\langle l, \sigma \rangle \xrightarrow{a(\overline{d})} \langle l', \sigma' \rangle$ generated in response to $a(\overline{d})$.

For a traversed EFSM edge we use the following match variables (with associated meaning):

event_type	is the event type a of the occurring event
event-pars	is the list $\overline{d}$ of parameters of the event
edge	is the name e
target_loc	is the target location l'
guard	is the guard expression g
assignments	is the set $\overline{u} := \overline{expr}$ of assignments
target_val	is the function from EFSM state variables to values, s.t. $val(u)$ is the value $\sigma'(u)$ of variable u just after the computation step.

Similarly, we also define *source_loc* for the source location and *source_val* for the value $\sigma(u)$ of variable u just before the computation step. To be able to express more interesting properties we also introduce a set of operations that can be used together with the match variables:

- lhs is a function to get the left hand side expression of an assignment. A left hand side expressions is always assumed to be a variable.
- rhs is a function to get the right hand side expressions of an assignment. The right hand side expression, $expr$, uses the vocabulary defined for the EFSM specification.
- $vars$ is a function such that $vars(Exp)$ returns a set with all variables found in Exp. Exp is a set that contains the result of applying rhs to each assignment in $assignments$, or a $guard$ expression.
- *affect* is a function such that $affect(A, Var_1, Var_2)$ returns the assignment it is being applied to, A, if $Var_1 \in vars(rhs(A)) \wedge Var_2 = lhs(A)$ otherwise the empty set is returned.
- map is a function such that $map(Fun, Set)$ applies the function, Fun on each element in the set Set and returns the set of the results.

With the match variables and operations above we define new functions that can be used as tests in the observer. In this paper, we shall make use of:

- $def(v)$, which is true iff the variable v is defined by the transition in the EFSM. This can be expressed as:

$$v \in map(lhs, assignments)$$

- $use(v)$, which is true iff the variable v is used (in a guard or assignment) by the transition in the EFSM. This can be expressed as:

$$v \in vars(map(rhs, assignments)) \vee v \in vars(guard)$$

- $da(v_1, v_2)$, which is true iff the variable v_1 is on the right hand side and variable v_2 is on the left hand side of the same assignment in the

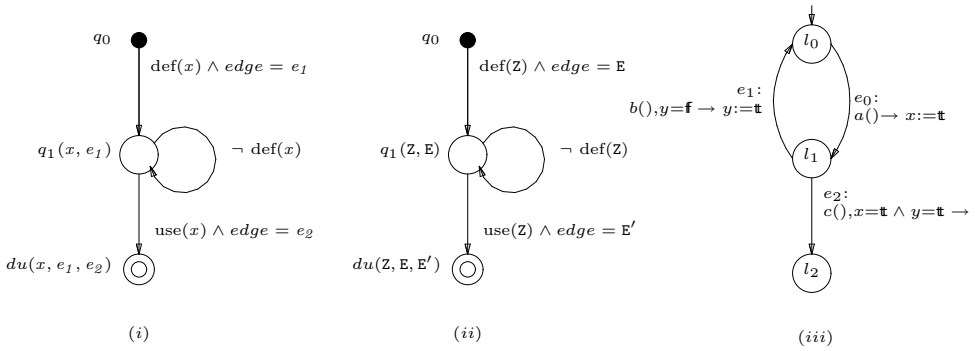


Figure 2: Examples of (i) observer monitoring definition (on edge e_1) and use (on edge e_2) of variable x, (ii) a parameterized observer, and (iii) a simple EFSM.

EFSM specification. The function can intuitively be understood to be true if v_1 directly affects v_2. This can be expressed as:

$$map(affect(v_1, v_2), assignments) \neq \emptyset$$

Example 2 *The (non-parameterized) observer in Figure 2(i) specifies definition-use pair coverage for a specific variable m, and specific edges e_1 and e_2. Figure 2(ii) shows a corresponding (parameterized) observer that specifies definition-use pair coverage for* any *EFSM variable* Z*, and EFSM edges* E *and* E′*. This is done by parameterizing the location q_1 with any variable and any edge, and the accepting location du with any variable and any two edges. The edges are parameterized in a similar way. For example, there is one observer edge from location $q_1(z, e)$ to location $du(z, e, e')$ for each EFSM variable z, and each pair e, e' of EFSM edges.*

3.2 How Observers Monitor Coverage Criteria

In test case generation or when monitoring test execution of a SUT, an observer observes the events of the SUT, and the computation steps of the EFSM. Reached accepting locations correspond to covered coverage items. We formally define the execution of an observer in terms of a composition between an EFSM and an observer, which has the form of a *superposition* of the observer onto the EFSM. Each state of this superposition consists of a state of the EFSM, together with a *set* of currently occupied observer locations.

Say that a predicate b on an observer edge is satisfied by a computation step $\langle l, \sigma\rangle \xrightarrow{a(\overline{d})} \langle l', \sigma'\rangle$ of an EFSM, denoted $\langle l, \sigma\rangle \xrightarrow{a(\overline{d})} \langle l', \sigma'\rangle \models b$ if b holds

for the event $a(\overline{d})$, the computation step $\langle l, \sigma\rangle \xrightarrow{a(\overline{d})} \langle l', \sigma'\rangle$, and the edge $e : l \xrightarrow{a(\overline{w}), g \to \overline{u} := \overline{expr}} l'$ from which the computation step is derived.

Formally, the superposition of an observer (Q, q_0, Q_f, B) onto an EFSM $\langle L, l_0, \overline{v}, E\rangle$ is defined as follows.

- *States* are of the form $\langle\langle l, \sigma\rangle \parallel \mathcal{Q}\rangle$, where $\langle l, \sigma\rangle$ is a state of the EFSM, and $\mathcal{Q}$ is a set of locations of the observer.
- The *initial state* is the tuple $\langle\langle l_0, \sigma_0\rangle \parallel \{q_0\}\rangle$, where $\langle l_0, \sigma_0\rangle$ is the initial state of the EFSM, and q_0 is the initial location of the observer.
- A *computation step* is a triple $\langle\langle l, \sigma\rangle \parallel \mathcal{Q}\rangle \overset{a(\overline{d})}{\rightsquigarrow} \langle\langle l', \sigma'\rangle \parallel \mathcal{Q}'\rangle$ such that $\langle l, \sigma\rangle \xrightarrow{a(\overline{d})} \langle l', \sigma'\rangle$ and

$$\mathcal{Q}' = \left\{ q' \mid q \xrightarrow{b} q' \text{ and } q \in \mathcal{Q} \text{ and } \langle l, \sigma\rangle \xrightarrow{a(\overline{d})} \langle l', \sigma'\rangle \models b \right\}$$

- A state $\langle\langle l, \sigma\rangle \parallel \mathcal{Q}\rangle$ of the superposition *covers* the coverage item represented by the location $q_f \in Q_f$ if $q_f \in \mathcal{Q}$.

Note that the way the set $\mathcal{Q}$ is updated essentially results in an (on-the-fly) subset construction of the parameterised observer. Initially, $\mathcal{Q}$ contains only the initial observer location q_0. In the subsequent computation steps, $\mathcal{Q}$ contains the set of all occupied observer locations, representing already covered and partially covered coverage items. In each computation step, the set of occupied observer locations $\mathcal{Q}'$ is obtained by generating all possible successors to the locations in $\mathcal{Q}$, i.e. all q' such that there exists a $q \in \mathcal{Q}$ and an edge $q \xrightarrow{b} q' \in B$ with b satisfied by the computation step of the EFSM.

Recall that both the initial and all accepting observer locations have implicit self-loops with predicate *true*. This means that in the superposition of the observer onto an EFSM, the initial observer location q_0 is always occupied and all reached accepting observer locations (representing covered coverage items) are guaranteed to remain in $\mathcal{Q}$. The fact that q_0 is always occupied can be intuitively understood as allowing for the observer to non-deterministically start monitoring an EFSM (or a SUT) at *any* computation step of an run (or at any point during test execution).

Example 3 *If the observer in Figure 2(ii) is superposed onto the EFSM in Figure 2(iii), the following computation steps can be taken* $\langle\langle l_0, \{x{=}\mathtt{ff}, y{=}\mathtt{ff}\}\rangle \parallel \{q_0\}\rangle \overset{a()}{\rightsquigarrow} \langle\langle l_1, \{x{=}\mathtt{tt}, y{=}\mathtt{ff}\}\rangle \parallel \{q_0, q_1(x, e_0)\}\rangle \overset{b()}{\rightsquigarrow} \langle\langle l_0, \{x{=}\mathtt{tt}, y{=}\mathtt{tt}\}\rangle \parallel \{q_0, q_1(x, e_0), q_1(y, e_1)\}\rangle \overset{a()}{\rightsquigarrow} \langle\langle l_1, \{x = \mathtt{tt}, y = \mathtt{tt}\}\rangle \parallel \{q_0, q_1(x, e_0), q_1(y, e_1)\}\rangle \overset{c()}{\rightsquigarrow} \langle\langle l_2, \{x{=}\mathtt{tt}, y{=}\mathtt{tt}\}\rangle \parallel \{q_0, q_1(x, e_0), q_1(y, e_1), du(x, e_0, e_2), du(y, e_1, e_2)\}\rangle$. *Thus, the two possible definition-use pairs are covered.*

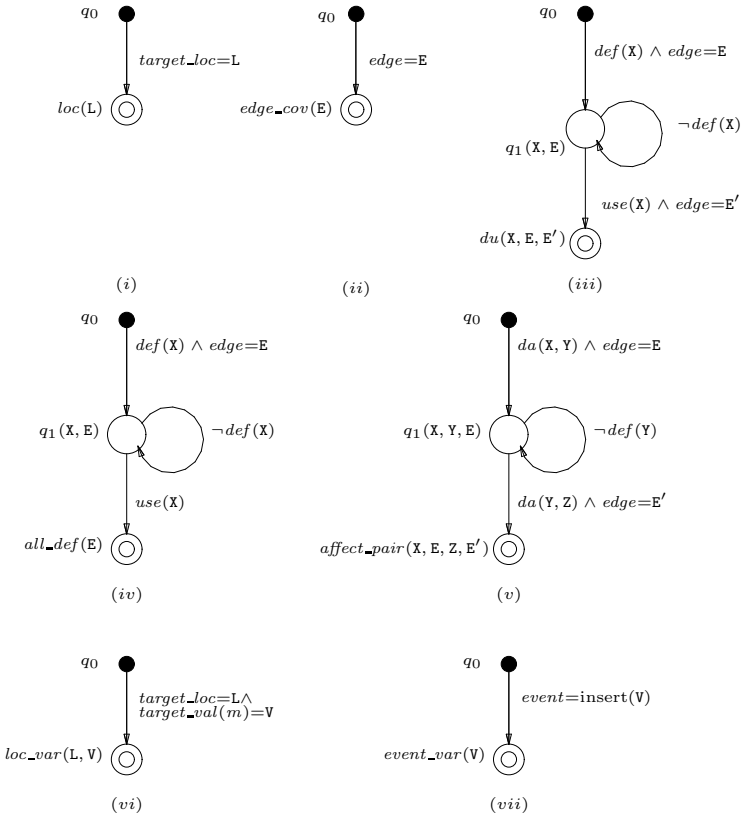


Figure 3: Seven examples of coverage criteria expressed as observers.

3.3 Examples of Observers

Figure 3 shows observers specifying a number of coverage criteria described in the literature [CPRZ89].

The *all-locations* coverage criteria is specified by the observer shown in Figure 3(i), where the parameter L is any location in an EFSM. If the observer is superposed onto the EFSM of Figure 1, we have that L = $\{IDLE, BUSY\}$ and the edge of the parameterized observer represents two edges, one guarded by $target\_loc = IDLE$ with target location $loc(IDLE)$ $target\_loc(BUSY)$, and the other guarded by $target\_loc = BUSY$ with target location $loc(BUSY)$. The set of possible coverage items is thus $\{loc(IDLE), loc(BUSY)\}$.

The *all-edges* coverage observer in Figure 3(ii) is similar to the all-location coverage observer. The edges of the EFSM in Figure 1 is E=$\{e_1, \ldots, e_5\}$, and thus the set of possible coverage items when the observer is superposed onto the EFSM is $\{ edge\_cov(e_i) \mid e_i \in \mathtt{E} \}$.

The *all-definition use-pairs* (all-uses [CPRZ89]) coverage observer in Fig-

ure 3(iii) has an accepting location $du(\mathtt{X}, \mathtt{E}, \mathtt{E}')$, where X is a variable name, E is an edge on which X is defined, and E′ an edge on which X is used. Variable X may not be redefined in the trace between E and E′. If the observer is superposed onto the EFSM the complete set of coverage items is $\{du(m, e_1, e_1),\ du(m, e_1, e_2),\ du(m, e_1, e_4),\ du(m, e_2, e_1),\ du(m, e_2, e_2),\ du(m, e_2, e_4),\ du(m, e_2, e_5)\}$. The definition-use pair $du(m, e_1, e_5)$ can not be covered since m is always redefined on edge e_2 in between e_1 and e_5.

The *all-definitions* coverage observer of Figure 3(iv) is similar to the all-definition use-pairs coverage except that only the defining edges are required to be covered. When the observer is superposed with the EFSM in Figure 1 the set of accepting locations is $\{all\_def(e_1), all\_def(e_2)\}$.

The *all affect-pairs* (Nafos' required k-Tuples [CPRZ89]) coverage observer shown in Figure 3(v) accepts whenever a variable x affects a variable z via another variable y. In this case we require that x directly affects y which, without redefinition, directly affects z. No such affect pairs are possible in the EFSM of Figure 1.

The *context coverage* criteria observer in Figure 3(vi) covers all values of a given variable m. We use $target\_val(m)$, to denote the value of m at the target *EFSM-state*. The observer has an accepting location $loc\_var(\mathtt{L}, \mathtt{V})$, where V is the value domain of variable m. E.g. $loc\_var(IDLE, 0)$ and $loc\_var(BUSY, 1)$ are accepting locations. The observer in Figure 3(vii) is similar, but covers the possible values the event parameter at transitions labelled with the event insert(x).

4 Test Case Generation

4.1 Algorithms

At test case generation, we use the superposition of an observer onto an EFSM, and views the test case generation problem as a search exploration problem. To cover a coverage item q_f is then the problem of finding a trace

$$tr = \langle\langle l_0, \sigma_0\rangle \parallel \{q_0\}\rangle \overset{a(\overline{d})}{\rightsquigarrow} \ldots \overset{a'(\overline{d}')}{\rightsquigarrow} \langle\langle l, \sigma\rangle \parallel \mathcal{Q}\rangle \text{ such that } q_f \in \mathcal{Q}$$

We will use $\omega(tr) = a(\overline{d}) \ldots a'(\overline{d}')$ to denote the *word* of the trace tr, or just ω whenever tr is clear from the context. In general, a single trace tr may cover several accepting locations of the observer. We say that the trace tr covers n accepting observer states if there are n accepting states in $\mathcal{Q}$, and we use $|Q_f \cap \mathcal{Q}|$ to denote the number of accepting states in $\mathcal{Q}$.

We are now ready to present the test case generation algorithm. We shall limit the presentation to an algorithm generating a single trace. The same technique can be used to produce sets of traces to cover many coverage items. Alternatively, the EFSM model can be annotated with edges that reset the

```
PASS:= ∅, MAX := 0, ω_max := ω_0
WAIT:= {⟨⟨s_0 ‖ {q_0}⟩, ω_0⟩}
while WAIT≠ ∅ do
   select ⟨⟨s ‖ Q⟩, ω⟩ from WAIT
   if |Q_f ∩ Q| > MAX then
      ω_max := ω, MAX := |Q_f ∩ Q|
   if for all ⟨s ‖ Q'⟩ in PASS: Q ⊈ Q' then
      add ⟨s ‖ Q⟩ to PASS
      for all ⟨s'' ‖ Q''⟩
         such that ⟨s ‖ Q⟩ ⇝^a ⟨s'' ‖ Q''⟩:
         add ⟨⟨s'' ‖ Q''⟩, ωa⟩ to WAIT
return ω_max and MAX
```

Figure 4: An abstract breadth-first search exploration algorithm for test case generation.

EFSM to its initial state. A generated trace can then be interpreted as a set of test cases separated by the reset edges [HLN+04].

An abstract algorithm to compute test case is shown in Figure 4. To improve the presentation, we use s to denote a system of the form $\langle l, \sigma \rangle$, s_0 to denote the initial system state $\langle l_0, \sigma_0 \rangle$, and a to denote an input action $a(\overline{d})$. The algorithm computes the maximum number of coverage items that can be visited (MAX), and returns a trace with maximum coverage (ω_{max}). The two main data structures WAIT and PASS are used to keep track of the states waiting to be explored, and the states already explored, respectively.

Initially, the set of already explored states is empty and the only state waiting to be explored is the extended state $\langle \langle s_0 \parallel \{q_0\} \rangle, \omega_0 \rangle$, where ω_0 is the empty trace. The algorithm then repeatedly examines extended states from WAIT. If a state $\langle s \parallel \mathcal{Q} \rangle$ found in WAIT is included in a state $\langle s \parallel \mathcal{Q}' \rangle$ in PASS, then obviously $\langle s \parallel \mathcal{Q} \rangle$ does not need to be further examined. If not, all successor states reachable from $\langle s \parallel \mathcal{Q} \rangle$ in one computation step are put on WAIT, with their traces extended with the input action of the computation step from which they are generated. The state $\langle s \parallel \mathcal{Q} \rangle$ is saved in PASS. The algorithm terminates when WAIT is empty

The variables ω_{max} and MAX are initially set to the empty trace and 0, respectively. They are updated whenever an extended state is found in WAIT which covers a higher number of coverage items than the current value of MAX. Throughout the execution of the algorithm, the value of MAX is the maximum number of coverage items that have been covered by a single trace, and ω_{max} is one such trace. When the algorithm terminates, the two values MAX and ω_{max} are returned.

4.2 Bitvector Implementation

In order to efficiently represent and manipulate the set $\mathcal{Q}$ of observer locations we shall use bitvector analysis [KSV96]. Let the set $\mathcal{Q}$ be represented by a bitvector where each bit represents an observer location q'. Then each bit is updated by the following function

$$f_{q'}(q') = \bigvee_{\langle b,q\rangle \in\ in(q')} q \wedge b$$

where $in(q') = \{\ \langle b, q\rangle \mid q \xrightarrow{b} q' \in B\ \}$ is the set of pairs of predicates b and source locations q of the edges ingoing to the location q'. That is, given a state of the superposition $\langle\langle l, \sigma\rangle \parallel \mathcal{Q}\rangle$ and an EFSM-transition $\langle l, \sigma\rangle \xrightarrow{a(\overline{d})} \langle l', \sigma'\rangle$ the bit representing q' is set to 1 if there is an observer edge $q \xrightarrow{b} q' \in B$, such that $q \in \mathcal{Q}$ and $\langle l, \sigma\rangle \xrightarrow{a(\overline{d})} \langle l', \sigma'\rangle \models b$. Otherwise the bit representing q' is set to 0. It should be obvious that this corresponds precisely to the semantics of an observer superposed onto an EFSM, described in Section 3.2.

Example 4 *When the observer in Figure 2(ii) is superposed onto the EFSM in Figure 2(iii), we have* $\mathtt{E} = \mathtt{E}' = E = \{e_0, e_1, e_2\}$ *and* $\mathtt{Z} = \overline{v} = \{x, y\}$. *Thus, we have that*

$$Q = \left\{\ q_0\ \right\} \cup \left\{\ q_1(z, e_a) \mid z \in \overline{v} \wedge e_a \in E\right\} \cup \left\{\ du(z, e_a, e_b) \mid z \in \overline{v} \wedge e_a, e_b \in E\right\}$$

Any enumeration of the set can be used as index in the bitvector. As the observer has three locations with parameters we get three types of bitvector functions:

$$\begin{aligned}
f_{q_0}(q_0) &= q_0 \wedge \mathtt{tt} && (1)\\
f_{q_1(v_i,e_j)}(q_1(v_i, e_j)) &= (\ q_0 \wedge def(v_i) \wedge (edge = e_j)\ ) \vee \\
&\quad (\ q_1(v_i, e_j) \wedge \neg def(v_i)\ ) && (2)\\
f_{du(v_i,e_j,e_k)}(du(v_i, e_j, e_k)) &= (\ q_1(v_i, e_j) \wedge use(v_i) \wedge (edge = e_k)\ ) \vee \\
&\quad (\ du(v_i, e_j, e_k) \wedge \mathtt{tt}\ ) && (3)
\end{aligned}$$

There is one function of type (1), six of type (2), and 18 of type (3). Note that (1) is always true and that (3) will remain true once it becomes true, due to implicit self-loops in these locations.

4.3 Implementation Efforts

Some of the techniques presented in this paper have been implemented in a prototype version of the model-checking tool UPPAAL [LPY97], extended for test case generation [HP04]. The current implementation uses the bitvector implementation described above, but is limited to a number of predefined

coverage criteria. For a given coverage criteria (a set of) test cases can be generated from system specifications described as DIEOU-timed automata [HLN+04]. We are currently in progress with a larger case-study in collaboration with Ericsson where this tool will be applied.

We are also developing a tool operating on a subset of the functional language Erlang, also using the techniques presented in this paper. The tool will be applied in a case-study in collaboration with Mobile Arts.

5 Conclusions

We have presented a technique for specifying coverage criteria in a simple and flexible manner using observer automata with parameters. Observers have shown to be a flexible tool in model checking and run-time monitoring, and by this paper we have shown that they are a versatile tool for specifying coverage criteria for test case generation and test monitoring. In particular the parameterization mechanism, as used in this paper, allows a succinct specification of several standard generic coverage criteria. In this way, test case generation can be transformed into a reachability problem, which can be attacked by a standard state-space reachability tool.

In previous works, we have implemented special cases of this test case generation technique, using UPPAAL, indicating that the approach is practical. We are currently working on a general implementation of the observer concept, and plan to apply it in a larger case study.

References

[BL03] F. Bouquet and B. Legeard. Reification of executable test scripts in formal specification-based test generation: The java card transaction mechanism case study. In *FME 2003*, volume 2805 of *Lecture Notes in Computer Science*, pages 778–795. Springer–Verlag, 2003.

[CPRZ89] L. A. Clarke, A. Podgurski, D. J. Richardsson, and Steven J. Zeil. A formal evaluation of data flow path selection criteria. *IEEE Trans. on Software Engineering*, SE-15(11):1318–1332, November 1989.

[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* 13^{th} *Int. Conference on Testing of Communicating Systems(TestCom 2000)*. Kluwer Academic Publishers, 2000.

[dZ99] L. du Bousquet and N. Zuanon. An overview of Lutess, a specification-based tool for testing synchronous software. In *Proc.* 14^{th} *IEEE Intl. Conf. on Automated SW Engineering*, October 1999.

[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.

[FJJV97] J.-C. Fernandez, C. Jard, T. Jéron, and C. Viho. An experiment in automatic generation of test suites for protocols with verification technology. *Science of Computer Programming*, 29, 1997.

[GLM02] S. Gnesi, D. Latella, and M. Massink. Modular semantics for a UML statechart diagrams kernel and its extension to multicharts and branching time model-checking. *Journal of Logic and Algebraic Programming*, 51(1):43–75, 2002.

[HCL+03] H.S. Hong, S.D. Cha, I. Lee, O. Sokolsky, and H. Ural. Data flow testing as model checking. In *ICSE'03:* 25^{th} *Int. Conf. on Software Enginering*, pages 232–242, May 2003.

[HLN+04] A. Hessel, K. G. Larsen, B. Nielsen, P. Pettersson, and A. Skou. Time-Optimal Real-Time Test Case Generation using Uppaal. In A. Petrenko and A. Ulrich, editors, *Proc.* 3^{rd} *International*

Workshop on Formal Approaches to Testing of Software 2003 (FATES'03), volume 2931 of *Lecture Notes in Computer Science*, pages 136–151. Springer–Verlag, 2004.

[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 : 8^{th} International Conference, (TACAS'02)*, volume 2280 of *Lecture Notes in Computer Science*, pages 327–341. Springer–Verlag, 2002.

[Hol97] G.J. Holzmann. The model checker SPIN. *IEEE Trans. on Software Engineering*, SE-23(5):279–295, May 1997.

[HP04] A. Hessel and P. Pettersson. A test generation algorithm for real-time systems. In H-D. Ehrich and K-D. Schewe, editors, *Proc. of 4^{th} Int. Conf. on Quality Software*, pages 268–273. IEEE Computer Society Press, September 2004.

[HR02] K. Havelund and G. Rosu. Synthesizing monitors for safety properties. In J.-P. Katoen and P. Stevens, editors, *Tools and Algorithms for the Construction and Analysis of Systems : 8^{th} International Conference, (TACAS'02)*, volume 2280 of *Lecture Notes in Computer Science*, pages 324–356. Springer–Verlag, 2002.

[ITU99] ITU, Geneva. *ITU-T, Z.100, Specification and Description Language (SDL)*, November 1999.

[KSV96] J. Knoop, B. Steffen, and J. Vollmer. Parallelism for free: Efficient and optimal bitvector analyses for parallel programs. *ACM Transactions on Programming Languages and Systems*, 18(3):268–299, 1996.

[LBV02] D. Lugato, C. Bigot, and Y. Valot. Validation and automatic test generation on UML models: the AGATHA approach. In *Proc. 7^{th} Int. Workshop on Formal Methods for Industrial Critical Systems (FMICS 02), Electronic Notes in Theoretical Computer Science*, volume 66, 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.

[MA00] B. Marre and A. Arnould. Test Sequence Generation from Lustre Descriptions: GATEL. In *Proc. 15^{th} IEEE Intl. Conf. on Automated Software Engineering (ASE'00)*, Grenoble, 2000.

[Meu00] C. Meudec. ATGen: Automatic test data generation using constraint logic programming and symbolic execution. In *Proc.* 1^{st} *Intl. Workshop on Automated Program Analysis, Testing, and Verification*, Limerick, 2000.

[Muc97] S. S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann, 1997.

[NS03] Brian Nielsen and Arne Skou. Automated test generation from timed automata. *International Journal on Software Tools for Technology Transfer*, 5:59–77, 2003.

[Pre01] A. Pretschner. Classical search strategies for test case generation with constraint logic programming. In *Proc.* 1^{st} *International Workshop on Formal Approaches to Testing of Software 2001 (FATES'01)*, pages 47–60, Aalborg, Denmark, August 2001.

[RdBJ00] V. Rusu, L. du Bousquet, and T. Jéron. An approach to symbolic test generation. In *Int. Conf. on Integrating Formal Methods*, volume 1945 of *Lecture Notes in Computer Science*, pages 338–357. Springer–Verlag, 2000.

[SEG+98] M. Schmitt, A. Ek, J. Grabowski, D. Hogrefe, and B. Koch. Autolink - putting sdl-based test generation into practice. In *11th Int. Workshop on Testing of Communicating Systems (IWTCS'98)*, Tomsk, Russia, September 1998.

[VW86] M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proc.* 1^{st} *Int. Symp. on Logic in Computer Science*, pages 332–344, June 1986.

[Wol81] P. Wolper. Temporal logic can be more expressive. In *Proc.* 22^{th} *Annual Symp. Foundations of Computer Science*, pages 340–348, 1981.

Recent licentiate theses from the Department of Information Technology

2006-002 Anders Hessel: *Model-Based Test Case Selection and Generation for Real-Time Systems*

2006-001 Linda Brus: *Recursive Black-box Identification of Nonlinear State-space ODE Models*

2005-011 Björn Holmberg: *Towards Markerless Analysis of Human Motion*

2005-010 Paul Sjöberg: *Numerical Solution of the Fokker-Planck Approximation of the Chemical Master Equation*

2005-009 Magnus Evestedt: *Parameter and State Estimation using Audio and Video Signals*

2005-008 Niklas Johansson: *Usable IT Systems for Mobile Work*

2005-007 Mei Hong: *On Two Methods for Identifying Dynamic Errors-in-Variables Systems*

2005-006 Erik Bängtsson: *Robust Preconditioned Iterative Solution Methods for Large-Scale Nonsymmetric Problems*

2005-005 Peter Nauclér: *Modeling and Control of Vibration in Mechanical Structures*

2005-004 Oskar Wibling: *Ad Hoc Routing Protocol Validation*

2005-003 Magnus Ågren: *High-Level Modelling and Local Search*

2005-002 Hakan Zeffer: *Hardware-Software Tradeoffs in Shared-Memory Implementations*

2005-001 Jesper Wilhelmsson: *Efficient Memory Management for Message-Passing Concurrency — part I: Single-threaded execution*

2004-006 Stefan Johansson: *High Order Difference Approximations for the Linearized Euler Equations*

2004-005 Henrik Löf: *Parallelizing the Method of Conjugate Gradients for Shared Memory Architectures*

UPPSALA
UNIVERSITET

Department of Information Technology, Uppsala University, Sweden