Time-Optimal Real-Time Test Case Generation Using UPPAAL

Anders Hessel[1], Kim G. Larsen[2], Brian Nielsen[2], Paul Pettersson[1], and Arne Skou[2]

[1] Department of Information Technology, Uppsala University
P.O. Box 337, SE-751 05 Uppsala, Sweden
{hessel,paupet}@it.uu.se
[2] Department of Computer Science, Aalborg University
Fredrik Bajersvej 7E, DK-9220 Aalborg, Denmark
{kgl,bnielsen,ask}@cs.auc.dk

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 [4,16,18,10]. 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.

A. Petrenko and A. Ulrich (Eds.): FATES 2003, LNCS 2931, pp. 114–130, 2004.

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 [11,9,6,17,8,5,7,14,15]. In [5,8,17] test sequences are generated from a timed automata (TA) by applying variations of finite state machine (FSM) checking sequence techniques (see eg. [13]) 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 [9] and [11] 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, [9] 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 [11] 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 [6] 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. [15] 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 [1,19,10], 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., [10].

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.

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 [2,12]. 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 [17], we define the notion of deterministic,

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

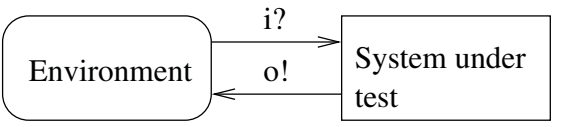

Fig. 1. Test Specification

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 $p \xrightarrow{\gamma} p'$ and $p \xrightarrow{\gamma} p''$ then $p' = p''$.
2. *(Weak) input enabled.* At any time any input action is enabled, i.e., whenever $p \xrightarrow{d}$ 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 $p \xrightarrow{\alpha}$ and $p \xrightarrow{\beta}$ then $\alpha = \beta$.
4. *Output urgency.* When an output (or τ) is enabled, it will occur immediately, i.e., whenever $p \xrightarrow{\alpha}, \alpha \in \mathcal{O} \cup \{\tau\}$ then $p \not\xrightarrow{d}, d \in \mathbb{R}_{\geq 0}$.

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 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.

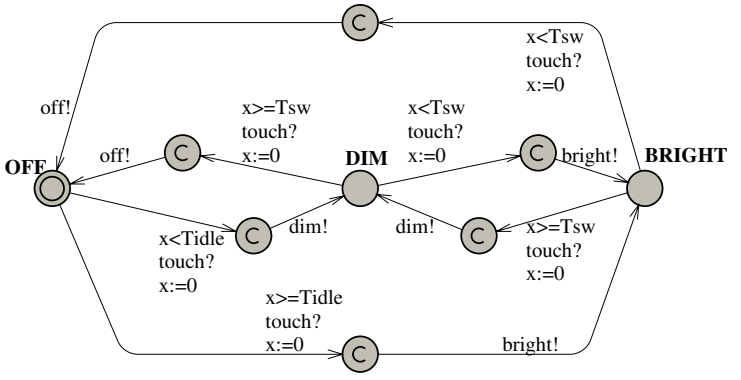

Fig. 2. Light Controller

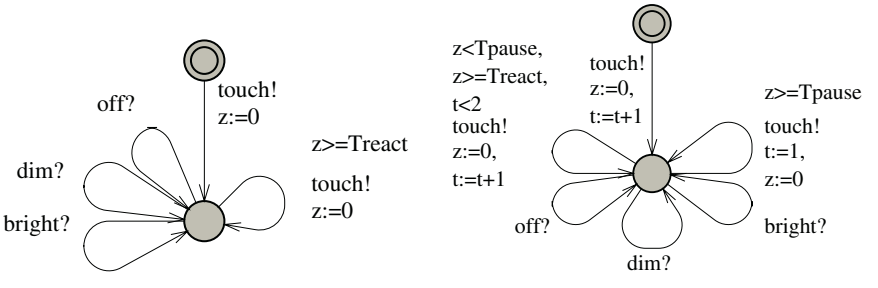

Fig. 3. Two possible environment models for the simple light switch

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 $\mathtt{t}$), 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:

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

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).

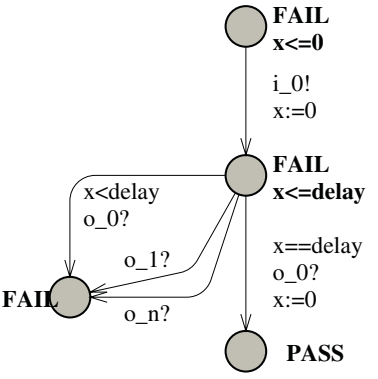

Fig. 4. Test case automaton for the sequence $i_0! \cdot \mathit{delay} \cdot o_0?$

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

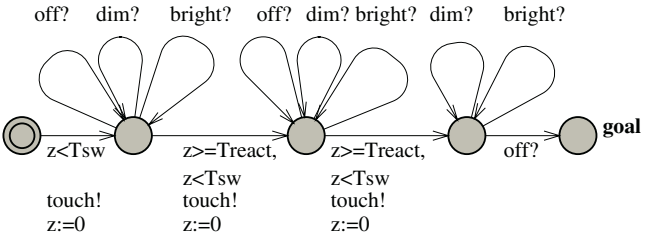

Fig. 5. Test Environment for TP2

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 $0 \cdot touch! \cdot 0 \cdot dim? \cdot 0 \cdot touch! \cdot 0 \cdot bright? \cdot 0 \cdot touch! \cdot 0 \cdot off?$.

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

[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.

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 $e_i := \mathbf{true}$; 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: $0 \cdot touch! \cdot 0 \cdot dim? \cdot 0 \cdot touch! \cdot 0 \cdot bright? \cdot 0 \cdot touch! \cdot 0 \cdot off? \cdot 20 \cdot touch! \cdot 0 \cdot bright? \cdot 4 \cdot touch! \cdot 0 \cdot dim? \cdot 4 \cdot touch! \cdot 0 \cdot off?$.

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 $if(v_d \neq \mathbf{false})then\ du[v_d, e_j] := \mathbf{true}$. 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 3(a). 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

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).

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(a). 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 6(b) reacts to `touch` events by switching the light on to the previous light level OL, or off. The user is modeled in Figure 6(c).

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.

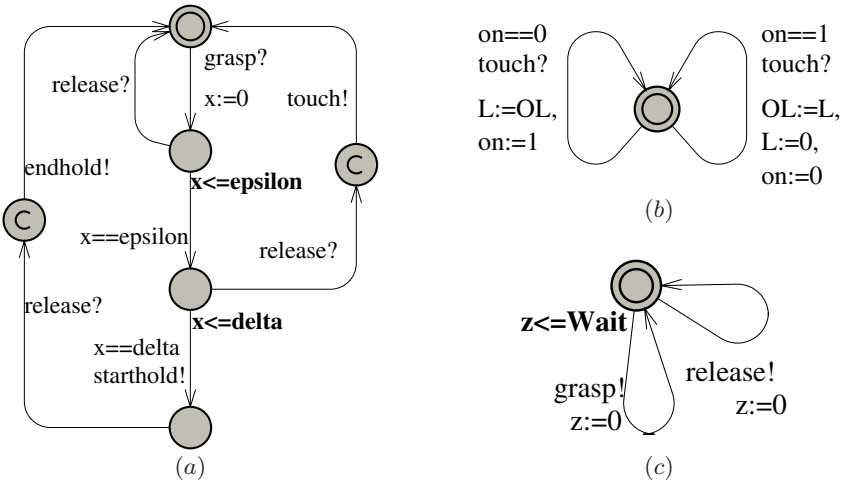

Fig. 6. Interface Automaton (a), Switch Automaton (b), and User Automaton (c)

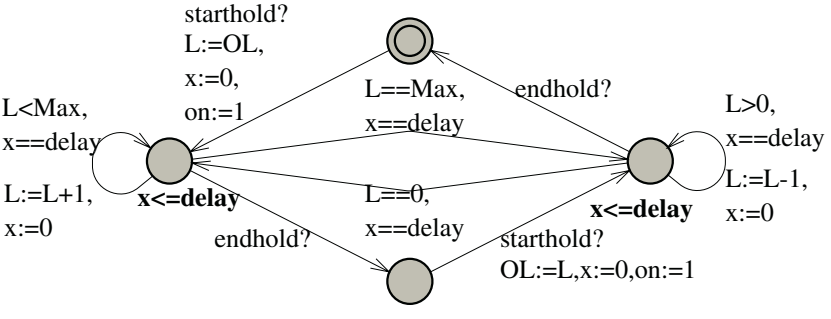

Fig. 7. Dimmer Automaton

Table 8 shows the optimal execution times (in time units) for test suites 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 9, 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

Table 8. Optimal execution time and suite length for various coverage criteria

	Impatient		Patient	
Coverage Criterion	Execution time	Suite length	Execution 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. Cost of obtaining edge coverage of the TSS with increasing light levels

	Impatient			Patient		
Levels	Execution time	Generation time(s)	Memory usage (MB)	Execution time	Generation time (s)	Memory usage(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

units), the reason being that the light 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 is 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 [3] 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

Table 10. Results for the Philips audio protocol

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

	First Solution			Optimal Solution	
Search order	Execution time	Generation time (s)	Memory usage (MB)	Generation time	Memory usage (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

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 10 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 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 11. 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 11 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. We are working on loosening the output urgency requirement. It may also be interesting to formulate coverage criteria that considers clock constraints.

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 ignore these bits when possible, and that takes advantage of them to prune states with "less" coverage.

References

1. Alfred V. Aho, Anton T. Dahbura, David Lee, and M. Ümit 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.

2. Gerd Behrmann, Ansgar Fehnker, Thomas Hune, Kim G. Larsen, Paul Pettersson, and Judi Romijn. Efficient Guiding Towards Cost-Optimality in UPPAAL. In T. Margaria and W. Yi, editors, *Proc. of TACAS 2001*, number 2031 in Lecture Notes in Computer Science, pages 174–188. Springer–Verlag, 2001.
3. 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*, number 863 in Lecture Notes in Computer Science, 1994.
4. Marius Bozga, Jean-Claude Fernandez, Lucian Ghirvu, Claude Jard, Thierry Jéron, Alain Kerbrat, Pierre Morel, and Laurent Mounier. Verification and Test Generation for the SSCOP Protocol. *Science of Computer Programming*, 36(1):27–52, 2000.
5. Rachel Cardell-Oliver. Conformance Testing of Real-Time Systems with Timed Automata. *Formal Aspects of Computing*, 12(5):350–371, 2000.
6. R. Castanet, Ousmane Koné, and Patrice 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.
7. Duncan Clarke and Insup Lee. Automatic Test Generation for the Analysis of a Real-Time System: Case Study. In *3rd IEEE Real-Time Technology and Applications Symposium*, 1997.
8. Abdeslam En-Nouaary, Rachida Dssouli, Ferhat Khendek, and A. Elqortobi. Timed Test Cases Generation Based on State Characterization Technique. In *19th IEEE Real-Time Systems Symposium (RTSS'98)*, pages 220–229, December 2–4 1998.
9. Teruo Higashino, Akio Nakata, Kenichi Taniguchi, and Ana R. Cavalli. Generating Test Cases for a Timed I/O Automaton Model. In Gyula Csopaki, Sarolta Dibuz, and Katalin Tarnay, editors, *Testing of Communicating Systems: Method and Applications, IFIP TC6 12th International Workshop on Testing Communicating Systems (IWTCS), September 1-3, 1999, Budapest, Hungary*, volume 147 of *IFIP Conference Proceedings*, pages 197–214. Kluwer, 1999.
10. Hyoung Seok Hong, Insup Lee, Oleg Sokolsky, and Hasan Ural. A Temporal Logic Based Theory of Test Coverage and Generation. In J.-P. Katoen and P. Stevens, editors, *TACAS 2002*, pages 327–341. Kluwer Academic Publishers, April 2002.
11. Ahmed Khoumsi. A method for testing the conformance of real-time systems. In Werner Damm and Ernst-Rüdinger Olderog, editors, *IEEE International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems. (FTRTFT*, volume 2469 of *LNCS*. Sprnger-Verlag, september 2002.
12. Kim G. Larsen, Gerd Behrmann, Ed Brinksma, Ansgar Fehnker, Thomas Hune, Paul Pettersson, and Judi 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*, number 2102 in Lecture Notes in Computer Science, pages 493–505. Springer–Verlag, 2001.
13. David Lee and Mihalis Yannakakis. Principles and Methods of Testing Finite State Machines—A Survey. *Proceedings of the IEEE*, 84(8):1090–1123, august 1996.
14. Dino Mandrioli, Sandro Morasca, and Angelo Morzenti. Generating Test Cases for Real-Time Systems from Logic Specifications. *ACM Transactions on Computer Systems*, 13(4):365–398, 1995.
15. Brian Nielsen and Arne Skou. Automated Test Generation from Timed Automata. *International Journal on Software Tools for Technology Transfer (STTT)*, 4, 2002. Digital Object Identifier (DOI) 10.1007/s10009-002-0094-1. To Appear.
16. Jan 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.

17. J. Springintveld, F. Vaandrager, and P.R. D'Argenio. Testing Timed Automata. *Theoretical Computer Science*, 254(1-2):225–257, March 2001.
18. 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.
19. M. Ümit Uyar, Marius A. Fecko, Adarsphal S. Sethi, and Paul 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.