Time-Optimal Test Cases for Real-Time Systems

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. E-mail: {`hessel,paupet`}`@it.uu.se`.
[2] Department of Computer Science, Aalborg University, Fredrik Bajersvej 7E,
DK-9220 Aalborg, Denmark. E-mail: {`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 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.

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 [6,14,16,10]. However, little is still known in the context of real-time systems, and few proposals exist that deals explicitly and systematically with testing real-time properties [15,9,7,8,12,13]. A principle problem is that a very large number of test cases (generally infinitely many) can be generated from even the simplest models. The addition of real-time adds another source of explosion, i.e. *when* to stimulate the system and expect response.

In this paper we demonstrate 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. 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 changes. 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 a some state, also has a high likelihood of detecting errors, because this is a stressful situation for the SUT to handle. Most other work, e.g [1,17], 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.

We propose a new technique for automatically generating time optimal test cases and test suites for embedded real time systems. 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. The fact that the SUT is a black box means that communication with the SUT only takes place via a well defined set of observable actions which implies limited observability and controllability. The required behavior is specified using Uppaal style timed automata. The fastest diagnostic trace facility of the Uppaal model checking tool is used to generate time optimal test sequences.

The test cases can either be generated using manually formulated test purposes or automatically from several kinds of coverage criteria—such as transition or location coverage–of the timed automata model. Even coverage based test suites are guaranteed to be time optimal in the sense the total time required to execute the test sequences in the suite (and the intermediate resets) is minimal. The main contributions of the paper are:

- Definition of a subclass of timed automata from which the diagnostic traces of Uppaal can be used as test cases.
- Application of time optimal reachability analysis algorithms to the context of test case generation.
- A technique to generate time optimal covering test suites for three important coverage criteria.
- Experimental evidence in that the proposed technique has practical merits.

The rest of the paper is organized as follows: in the next section we introduce a framework for testing real-time systems based on a testable subclass of timed automata. In Section 3 and 4 we describe how to encode test purposes and test criteria, and report experimental results respectively. In Section 5 we conclude the paper and discuss future work.

2 Timed Automata and Testing

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

2.1 Testable Timed Automata

The model used in this paper is networks of timed automata [2] with a few restriction to ensure testability.

Let X be a set of non-negative real-valued variables called *clocks*, and Act a set of actions and co-actions (denoted $a!$ and $a?$) and the non-synchronising 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 the form $x := c$, where $x \in X$, $c \in \mathbb{N}$, and $\bowtie \in \{\leq, <, =, \geq\}$[1]. A *timed automaton* 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, $I : L \rightarrow \mathcal{G}(X)$ 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 $\ell \xrightarrow{g,a,u} \ell'$ iff $(\ell, g, a, u, \ell') \in E$.

The semantics of a timed automaton 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 $(\ell, \sigma) \xrightarrow{a} (\ell', \sigma')$ correspond to execution of edges (ℓ, g, a, u, ℓ') 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.

A *network of timed automata* $\mathcal{A}_1 \parallel \cdots \parallel \mathcal{A}_n$ over (Act, X) is defined as the parallel composition of n timed automata 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 $a!$).

To ensure testability, certain semantic restrictions turn out to be required. Following similar restrictions in [15], we define the notion of deterministic, input enabled and output urgent timed automata, DIEOU-TA, as follows:

1. *Determinism.* For a given state p and label l, all transitions of form $p \xrightarrow{l}$ lead to the same state.
2. *Input enableness.* In any state, any input action is enabled.
3. *Output uniqueness.* Each state p has at most *one* out action, i.e. $p \xrightarrow{a!}$, $p \xrightarrow{b!}$ implies $a = b$.
4. *Output urgency.* When an output (or τ) is enabled, it will occur immediately, i.e. time is not allowed to pass when $p \xrightarrow{a!}$ (or $p \xrightarrow{\tau}$).

---

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

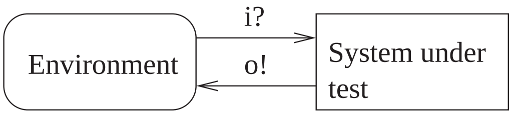


Figure1. Test Specification.

2.2 Testing Timed Automata

We assume that the test specification is given as a closed network of timed automata that can be partitioned into one subnetwork modelling the behavior of the SUT, and one modelling the behavior of its environment (ENV), as shown in Figure 1. Often the SUT operates in a specific environment, in case it is only necessary to establish correctness under the (modelled) environment assumptions; otherwise the environment model can be replaced with a unconstrained environment allowing all possible interaction sequences.

We assume that the tester can take the place of the environment and control the SUT via a distinguished set of observable input ($\mathcal{I}$) and output actions ($\mathcal{O}$), $Act = \mathcal{I} \cup \mathcal{O}$. For the SUT to be testable the subnetwork modelling 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. edges and locations). This is precisely ensured by making the assumption that the model of the system under test satisfy the restrictions of DIEOU.

Example 1. We use the simple light switch controller shown 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 $T_{idle} = 20$), 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 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 modelled setting T_{react} to a non-zero value. Figure 3(b) models a different user able to make two quick successive touches, but which then is required to pause for some time (to avoid cramp) $T_{pause} = 5$.

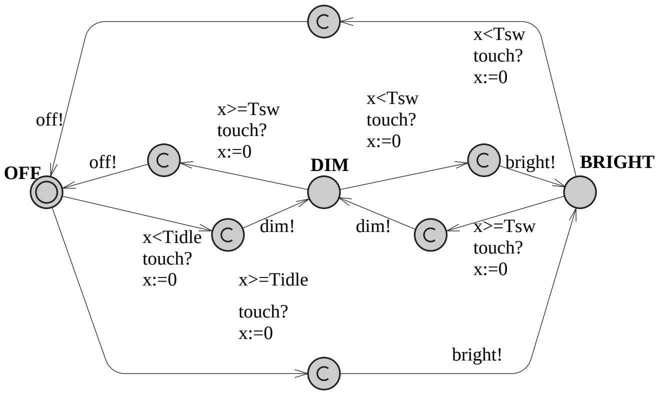


Figure2. Light Controller.

2.3 UPPAAL and Time-Optimal Reachability

Uppaal is a verification tool for a timed automata based modelling language [11]. Besides dense-time clocks, the tool supports both simple and complex data types like bounded integers and arrays as well as synchronisation via shared variables and actions. The specification language supports both safety and liveness properties.

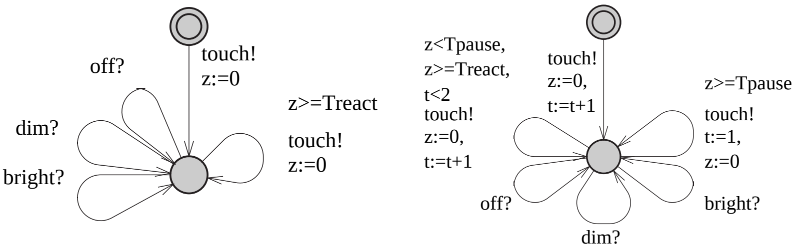
To produce test sequences, we shall make use of Uppaal's ability to generate diagnostic traces witnessing a posed safety property. Currently Uppaal support three options for diagnostic trace generation: *some trace* leading to the goal state, the *shortest trace* with the minimum number of transitions, and *fastest trace* with the shortest accumulated time delay. The underlying algorithm used for finding time-optimal traces is an extended version of Uppaal's symbolic on-the-fly reachability analysis algorithm, extended with ideas from the well-known A*-algorithm [3]. Hence to further improve performance it is possible to supply a heuristic function which, for all reachable symbolic states, gives a lower bound estimation of the remaining cost needed to reach a goal state.

2.4 From Diagnostic Traces to Test Cases

Let A be the timed automata network model of the SUT together with its inteded environment ENV. Consider a (concrete) diagnostic trace produced by Uppaal for a given reachability question on A. This trace will have the form:

$$(S_0, E_0) \xrightarrow{l_0} (S_1, E_1) \xrightarrow{l_1} (S_2, E_2) \xrightarrow{l_2} \cdots (S_n, E_n)$$

where S_i, E_i are states of the SUT and ENV, respectively, and l_i are either time-delays or synchronization (or internal) actions. The latter may be further



(a) (b)

Figure3. Two possible environment models for the simple light switch

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).

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.

To clarify the construction we may model the test case itself as a timed automaton A_λ for the test sequence λ. Locations in A_λ are labelled 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 i **pass** state if the SUT can execute all actions of λ. The construction is illustrated in Figure 4.

3 Test Generation

In this section we describe how to generate time-optimal test sequences from test purposes, and time-optimal test suites from coverage criteria.

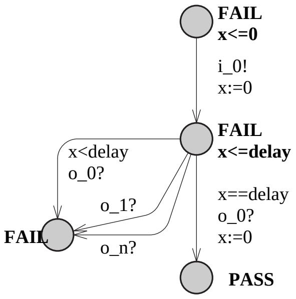


Figure4. Test case automaton for the sequence $i_0! \cdot delay \cdot o_0?$.

3.1 Single Purpose Test Generation

A common approach to the generation of test cases is to first manually formulate informally a set of test purposes and then to formalize them 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.

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.

The test purpose **TP1** can be formulated as a simple reachability property: `E<> LightController.bright` (i.e. eventually the `LightController` automaton enters location `bright`). Generating the *shortest* diagnostic trace results in the test sequence: $20 \cdot touch! \cdot bright?$. However, the *fastest sequence* satisfying the purpose is $0 \cdot touch! \cdot dim? \cdot 0 \cdot touch! \cdot bright?$.

Test purpose **TP2** can be formalized using the restricted environment model[2] in Figure 5 with the property `E<> tpEnv.goal`. The fastest test sequence is $0 \cdot touch! \cdot dim? \cdot 0 \cdot touch! \cdot bright? \cdot 0 \cdot touch! \cdot off?$.

---

[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.

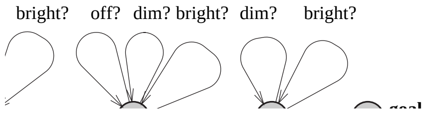


Figure5. Test Environment for TP2.

3.2 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 generalise 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 timed automata models.

Edge Coverage: A test sequence satisfies the *edge-coverage criterion* if, when executed on the model, it traverses every edge of the selected network 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 light switch in Figure 2 requires a bit-array of 12 elements. When the environment can touch arbitrary fast the generated fastest edge covering test sequence has accumulated execution time 28. The solution (there might be more traces with the same fastest execution time) generated by UPPAAL is:
EC: $0 \cdot \mathit{touch}! \cdot \mathit{dim}? \cdot 0 \cdot \mathit{touch}! \cdot \mathit{bright}? \cdot 0 \cdot \mathit{touch}! \cdot \mathit{off}? \cdot 20 \cdot \mathit{touch}! \cdot \mathit{bright}? \cdot 4 \cdot \mathit{touch}! \cdot \mathit{dim}? \cdot 4 \cdot \mathit{touch}! \cdot \mathit{off}?$.

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 $s_i := \textbf{true}$; 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 $\{\textbf{false}\} \cup \{1 \dots |E|\}$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 $\mathit{if}(v_d \neq \textbf{false})\mathit{then}\ du[v_d, e_j] := \textbf{true}$. 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.

3.3 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), or there might exist a covering set of test sequences for which the total time is shorter than for the fastest covering single test sequence. In these cases, the time-optimal test suite (i.e. the set of test sequences with shortest accumulated time) is needed to test the system. To generate time-optimal test suites, we shall introduce in the model *resets* that resets the model to its initial state, from which the test may continue to cover the remaining parts. The generated test is then interpreted as a test suite consisting of a set of test sequences separated by resets.

To introduce resets in the model, we allow the user to designate some locations as being resettable. Obviously, performing a reset in practice may take some time T_r (or other costs measured in time) that must be taken into consideration when generating time-optimal test sequences. Resettable 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 resettable location. Two reset-edges must then be added from ℓ to the initial location ℓ_0, i.e.,

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

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.

3.4 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 necessary to establish correctness of SUT only under the (modelled) environment assumptions. Therefore assumptions about the environment should be modelled explicitly, and will then be taken into account during test sequence generation.

4 Experiments

In the previous section we present techniques to compute time-optimal covering test suites. In the following we apply the presented technique to a version of Philips audio control protocol [5,4], frequently studied in the context of model checking.

We have created a DIEOU-TA model of the the protocol. The system consists of a sender component and a receiver component 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. To detect collisions the sender also checks 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 5 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 collision, and a sender buffer producing any legal input-bit sequence. The second row shows results for a receiver testing in an environment consiting 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

---

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

Criteria	E(μs)	G (s)	M (Kb)
EC_S	212350	2.2	9416
EC_R	18981	1.2	4984
$EC_{R,S}$	114227	129.0	331408

Table5. Results for the Philips audio-control protocol.

system. In all cases the time optimal covering test sequence could be computed in reasonable time.

5 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. The technique uses the time optimal reachability feature of UPPAAL. Using a version of Philips audio-control protocol, 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 removing the output urgency requirement. Without fundamental changes our technique can be applied to models that are output persistent only, meaning that outputs are allowed to appear at some unspecified time in an interval.

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

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. Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

3. 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.
4. Johan Bengtsson, W. O. David Griffioen, Kåre J. Kristoffersen, Kim G. Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. Automated analysis of an audio control protocol using uppaal. *Journal of Logic and Algebraic Programming*, 52–53:163–181, July-August 2002.
5. 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.
6. 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.
7. Rachel Cardell-Oliver. Conformance Testing of Real-Time Systems with Timed Automata. *Formal Aspects of Computing*, 12(5):350–371, 2000.
8. 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.
9. Abdeslam En-Nouaary, Rachida Dssouli, and Ferhat Khendek. 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.
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. Kim G. Larsen, Paul Pettersson, and Wang Yi. Uppaal in a Nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, October 1997.
12. 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.
13. 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.
14. Jan Peleska. Hardware/Software Integration Testing for the new Airbus Aircraft Families. In A. Wolis I. Schieferdecker, H. Knig, editor, *Testing of Communicating Systems XIV. Application to Internet Technologies and Services*, pages 335–351. Kluwer Academic Publishers, 2002.
15. J. Springintveld, F. Vaandrager, and P.R. D'Argenio. Testing Timed Automata. *Theoretical Computer Science*, 254(1-2):225–257, March 2001.
16. 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.
17. 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.