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
{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 UP-PAAL, *making it possible to express a variety of coverage criteria.*

1. Introduction

In [7], we have presented a technique for generating time-optimal test suites from timed automata specifications using UPPAAL [9]. 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 UP-PAAL.

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. Preliminaries

We will use a restricted type of timed automata [1], extended with finite domain variables, called DIEOU-TA [7] 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, $I : L \rightarrow \mathcal{G}(X)$ 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 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 $\sigma \in \mathbb{R}_{\geq 0}^X$ 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!).

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 [11], we define the notion of deterministic, input enabled and output urgent TA, DIEOU-TA [7], 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 $p \xrightarrow{\gamma} p'$ and $p \xrightarrow{\gamma} p''$ then $p' = p''$.

2. *(Weak) input enabled.* 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.* $\forall \alpha \in \mathcal{O} \cup \{\tau\}. \forall \beta \in \mathcal{O} \cup \mathcal{I} \cup \{\tau\}$ whenever $p \xrightarrow{\alpha}$ and $p \xrightarrow{\beta}$ then $\alpha = \beta$.

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

2.3. UPPAAL and Testing

UPPAAL [9] 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 [7] 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.

The technique presented in [7] 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 [6] 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 1. The algorithm explores symbolic

---

```
Pass:= ∅
Wait:= {((l₀, D₀), C₀)}
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.

states of the form (l, D), where D is a zone (or DBM [5]) 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. [3, 10] for a description of the symbolic semantics implemented in Up-paal). 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. [8]).

The algorithm in Figure 1 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 straightforward. 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 [7] 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 1 (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 1. The case of definition-use pair coverage is the same except that

```
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.

$F = \{\}$ 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 [3]. 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 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 [3] for more details.

In [3], it is also shown how the algorithm can be extended with a set of techniques inspired by branch and bound algorithms [2]. 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) C_s in a transition like $((\overline{l}, D), C) \rightsquigarrow_c ((\overline{l}_s, D_s), C_s)$. The input to the module is the coverage set C, a symbolic transition, and the new symbolic state, i.e. $\rightsquigarrow (\overline{l}_s, D_s)$. The example shown in the Figure 3 calculates C_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 C to C_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 $\rightsquigarrow (\overline{l}_s, D_s)$ and produces a bitvector $δ_i$ with the bits set that correspond to items covered by the given transition and state. The successor set C_s is created by *bitwise-or* of the old set C and $δ_i$ for all atomic objects. Thus, in the general case $C_s = C \cup (\bigcup_i δ_i)$.

An object in the atomic layer is created for each atomic coverage described by the search property $φ_c$. In case of
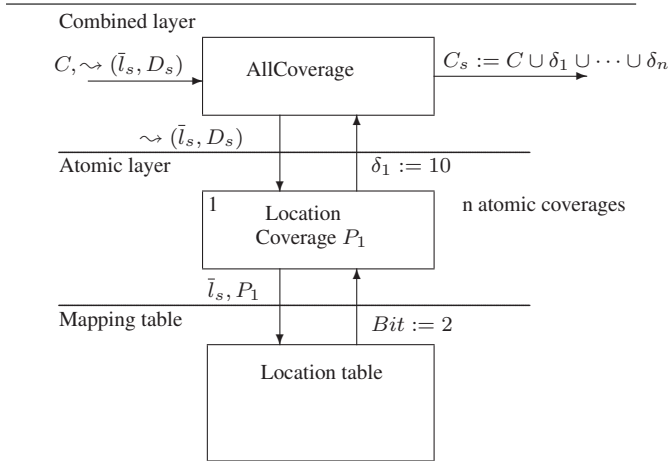
Figure 3. Architecture of the coverage module. Location coverage example instantiation.

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 C 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.

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 [4]. The protocol sends Manchester encoded bitstreams over a bus link, and detects collisions if two senders try to send that the same timepoint. We use the model presented by Bengtsson et.al. in [4]. 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 en-

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

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

vironment 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 $\unlhd$). 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.

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

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

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

[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, *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, number 2031 in Lecture Notes in Computer Science, pages 174–188. Springer–Verlag, 2001.

[4] Johan Bengtsson, W.O. David Griffioen, Kare J. Kristoffersen, Kim G. Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. Verification of an Audio Protocol with Bus Collision Using UPPAAL. In Rajeev Alur and Thomas A. Henzinger, editors, *Proc. of the 8th Int. Conf. on Computer Aided Verification*, number 1102 in Lecture Notes in Computer Science, pages 244–256. Springer–Verlag, July 1996.

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

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

[7] 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, *Proc. of 3rd International Workshop on Formal Approaches to Testing of Software*, number 2931 in Lecture Notes in Computer Science, pages 136–151. Springer–Verlag, 2003.

[8] Kim G. Larsen, Paul Pettersson, and Wang Yi. Diagnostic Model-Checking for Real-Time Systems. In *Proc. of Workshop on Verification and Control of Hybrid Systems III*, number 1066 in Lecture Notes in Computer Science, pages 575–586. Springer–Verlag, October 1995.

[9] 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.

[10] Fredrik Larsson, Kim G. Larsen, Paul Pettersson, and Wang Yi. Efficient Verification of Real-Time Systems: Compact Data Structures and State-Space Reduction. In *Proc. of the 18th IEEE Real-Time Systems Symposium*, pages 14–24. IEEE Computer Society Press, December 1997.

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