Timed Diagnostics for Reachability Properties

Stavros Tripakis

VERIMAG
Centre Équation, 2, rue de Vignate, 38610, Gières, France
E-mail: tripakis@imag.fr

Abstract. We consider the problem of computing concrete diagnostics for timed automata and reachability properties. Concrete means containing information both about the discrete state changes and the exact amount of time passing at each state. Our algorithm solves the problem in $O(l \cdot n^2)$ time, where l is the length of the diagnostic run and n the number of clocks. A prototype implementation in the tool KRONOS has been used to produce a counter-example in the claimed-to-be-correct version of the collision detection protocol of [HSLL97].

1 Introduction

When checking a system against a property, a simple yes/no answer is often not satisfactory. The term *diagnostics* is used for any kind of supplementary information (for instance, states, executions or sets of these) which helps the user understand why verification fails or succeeds. Diagnostics are important for the following reasons:

- Without them no confidence in the system's model can be gained. For instance, in case the property is not satisfied by the model, it might be that it is not the system which is wrong, but the modeling.
- Even if the model is correct, the fault of the system cannot be easily located without any guidance.

In the particular case of timed systems modeled as dense-time automata (TA) [ACD93,HNSY94], there is a need for *timed* diagnostics, that is, *concrete runs* in the TA semantics. These runs contain information both about the discrete state changes of the system, as well as the exact time delay between two discrete transitions. These delays can be essential to the understanding of a sample behavior of the system.

Since TA model-checking is based on *abstract* models rather than the concrete (i.e., semantic) one [DT98,BTY97], timed diagnostics cannot be generated directly. Until now, TA verification tools like KRONOS [DOTY96,BTY97] and UPPAAL [HSLL97] have been able to produce only abstract diagnostics, that is, sequences of the form $S_1 \rightarrow \cdots \rightarrow S_k$, where $S_1, ..., S_k$ are sets of states and $\rightarrow$ is some abstract transition relation between these sets (usually corresponding to discrete steps). Then, all that is known is that some concrete execution exists which corresponds to the abstract one. In particular, all information about delays between discrete steps is lost.

W.R. Cleaveland (Ed.): TACAS/ETAPS'99, LNCS 1579, pp. 59–73, 1999.
© Springer-Verlag Berlin Heidelberg 1999

In this paper we show how to compute timed diagnostics for TA with respect to reachability properties. Our technique is based on, first, finding an abstract execution sequence like the one above, and then extracting from it the concrete states and time delays. The complexity of our algorithm is $O(l \cdot n^2)$, where l is the length of the abstract sequence and n is the number of clocks of the TA.

We have implemented our algorithm in the tool KRONOS and used it to verify the case study presented in [HSLL97], The case study concerns an industrial protocol by BANG&OLUFSEN, aimed to ensure collision detection in a distributed audio/video environment. [HSLL97] present two versions of the protocol: the first one contains an error, claimed to be corrected in the second version. Surprisingly, we have found an error even in the "corrected" version of the protocol. Using our algorithm, we have obtained a timed counter-example showing how a collision can pass undetected.

2 Background

2.1 Clocks, bounds and polyhedra

Let R be the set of non-negative reals and $\mathcal{X} = \{x_1, ..., x_n\}$ be a set of variables in R, called *clocks*. An *$\mathcal{X}$-valuation* is a function $\mathbf{v} : \mathcal{X} \mapsto \mathsf{R}$. For some $X \subseteq \mathcal{X}$, $\mathbf{v}[X := 0]$ is the valuation $\mathbf{v}'$, such that $\forall x \in X \,.\, \mathbf{v}'(x) = 0$ and $\forall x \notin X \,.\, \mathbf{v}'(x) = \mathbf{v}(x)$. For every $\delta \in \mathsf{R}$, $\mathbf{v} + \delta$ (resp. $\mathbf{v} - \delta$) is a valuation such that for all $x \in \mathcal{X}$, $(\mathbf{v} + \delta)(x) = \mathbf{v}(x) + \delta$ (resp. $(\mathbf{v} - \delta)(x) = \mathbf{v}(x) - \delta$). Two valuations $\mathbf{v}$ and $\mathbf{v}'$ are called *c-equivalent*, for $c \in \mathsf{N}$, if for any clock x, either $\mathbf{v}(x) = \mathbf{v}'(x)$ or $\mathbf{v}(x) > c$ and $\mathbf{v}'(x) > c$.

A *bound* [Dil89] over $\mathcal{X}$ is a constraint of the form of the form $x_i \sim c$ or $x_i - x_j \sim c$, where $1 \leq i \neq j \leq n$, $\sim \in \{<, \leq, \geq, >\}$ and $c \in \mathsf{N} \cup \{\infty\}$. If we introduce a "dummy" clock variable x_0, taken to represent 0, then bounds can be uniformly written as $x_i - x_j \prec c$, where $0 \leq i \neq j \leq n$, $\prec \in \{<, \leq\}$ and $c \in \mathsf{Z} \cup \{\infty\}$. For example, $x_1 > 3$ can be written as $x_0 - x_1 < -3$. A bound $x_i - x_j \prec c$ is *stricter* than $x_i - x_j \prec' c'$ iff either $c < c'$ or $c = c'$ and $\prec = <, \prec' = \leq$. (By convention, $\delta < \infty$, for any real number δ.) For instance, $x_i - x_j < 3$ is stricter than $x_i - x_j \leq 3$, which is stricter than $x_i - x_j < 4$, and so on. For two bounds b and b', $\min(b, b')$ (resp. $\max(b, b')$) is b (resp. b') if b is stricter than b', b' (resp. b) otherwise. An $\mathcal{X}$-valuation $\mathbf{v}$ satisfies a bound $x_i - x_j \prec c$ iff $\mathbf{v}(x_i) - \mathbf{v}(x_j) \prec c$, where $\mathbf{v}(x_0)$ is assumed to be 0.

An *$\mathcal{X}$-polyhedron* ζ is a conjunction of $n \cdot (n+1)$ bounds:

$$\zeta = \bigwedge_{0 \leq i \neq j \leq n} x_i - x_j \prec_{i,j} c_{i,j}$$

We write $\zeta(i,j)$ for the pair $(\prec_{i,j}, c_{i,j})$. We also write $c_{max}(\zeta)$ for $\max\{|c_{i,j}| \mid 0 \leq i \neq j \leq n\}$ ($|c|$ is the absolute value of c if $c \in \mathsf{Z}$, and 0 if $c = \infty$).

An $\mathcal{X}$-polyhedron ζ defines a semantic entity, namely, the set of valuations satisfying all its bounds. If the bounds of ζ are unsatisfiable, ζ defines the empty

valuation set. More than one different $\mathcal{X}$-polyhedra might define the same valuation set, as is the case for $\zeta_1 : x \leq 2 \wedge y \leq 2 \wedge 0 \leq x - y \leq 2$ and $\zeta_2 : x \leq 2 \wedge y \leq 2 \wedge 0 \leq x - y \leq 3$. In the sequel, we consider only polyhedra that are in *canonical form*, that is, where all constraints are as tight as possible [1]. This is the case for ζ_1 above, but not for ζ_2.

Canonical form reduces semantic equality to syntactic equality, so that the valuation sets defined by ζ and ζ' are equal iff $\zeta(i,j)$ and $\zeta'(i,j)$ are identical, for all $0 \leq i,j \leq n$. Other semantic operations on polyhedra, such as intersection $\zeta \cap \zeta'$, inclusion $\zeta \subseteq \zeta'$, time successors $\nearrow \zeta = \{\mathbf{v} \mid \exists \delta \in \mathsf{R} \,.\, \mathbf{v} - \delta \in \zeta\}$ and clock reset $\zeta[X := 0] = \{\mathbf{v}[X := 0] \mid \mathbf{v} \in \zeta\}$ also have their correspondent syntactic transformations (the reader can see [Dil89,Yov93] for details of their implementation).

The above operations are used in section 3 for performing abstract reachability analysis and computing timed diagnostics. An important operation in this analysis is the one guaranteeing termination, separately presented below.

c-closure. Given an $\mathcal{X}$-polyhedron ζ and a natural constant c, the *c-closure* of ζ, denoted $\mathsf{close}(\zeta, c)$, is the greatest $\mathcal{X}$-polyhedron $\zeta' \supseteq \zeta$, such that for all $\mathbf{v}' \in \zeta'$ there exists $\mathbf{v} \in \zeta$ such that $\mathbf{v}$ and $\mathbf{v}'$ are c-equivalent. Intuitively, ζ' is obtained by ζ by "ignoring" all bounds which involve constants greater than c. An example is shown in figure 1.

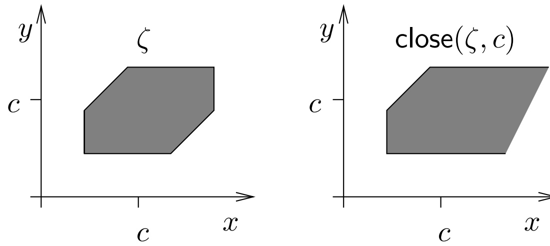

Fig. 1. An example of c-closure.

ζ is said to be *c-closed* if $\mathsf{close}(\zeta, c) = \zeta$.

Lemma 1. *For any constant c, there is a finite number of c-closed $\mathcal{X}$-polyhedra.*

The above result will be used in section 3.1 to prove termination of the reachability algorithm.

[1] A canonical form can be computed for any non-empty polyhedron, as shown in [Dil89], using an $O(n^3)$ algorithm, where n is the number of clocks.

2.2 Timed automata

A *timed automaton* (TA) [ACD93,HNSY94] is a tuple $A = (\mathcal{X}, Q, E)$, where:

- $\mathcal{X}$ is a finite set of clocks.
- Q is a finite set of *discrete states*.
- E is a finite set of *edges* of the form $e = (q, \zeta, X, q')$, where $q, q' \in Q$ are the *source* and *target* discrete states, ζ is a conjunction of atomic constraints on $\mathcal{X}$, called the *guard* of e, and $X \subseteq \mathcal{X}$ is a set of clocks to be reset.

Given an edge $e = (q, \zeta, X, q')$,we write $\mathsf{guard}(e)$ and $\mathsf{reset}(e)$ for ζ and X, respectively. Given a discrete state q, we write $\mathsf{out}(q)$ for the set of edges of the form $(q, _, _, _)$. Finally, we write $c_{max}(A)$ for the maximum of $c_{max}(\zeta)$, where ζ is a guard or an invariant of A.

States. A *state* of A is a pair $(q, \mathbf{v})$, where $q \in Q$ is a location, and $\mathbf{v}$ is an $\mathcal{X}$-valuation. Two states $(q, \mathbf{v}_1)$ and $(q, \mathbf{v}_2)$ are c-equivalent if $\mathbf{v}_1$ and $\mathbf{v}_2$ are c-equivalent.

Transitions. Consider a state $s = (q, \mathbf{v})$. We write $s + \delta$ instead of $(q, \mathbf{v} + \delta)$. A *timed transition* from s has the form $s \xrightarrow{\delta} s + \delta$, where $\delta \in \mathsf{R}$. $s + \delta$ is the *δ-successor* of s. A *discrete transition* with respect to an edge $e = (q, \zeta, X, q')$ such that $\mathbf{v} \in \zeta$ has the form $s \xrightarrow{e} s'$, where $s' = (q', \mathbf{v}[\mathsf{reset}(e) := 0])$. s' is the *e-successor* of s.

We write $s \xrightarrow{e}\xrightarrow{\delta} s'$ if, either $\delta = 0$ and $s \xrightarrow{e} s'$ is a discrete transition, or $\delta > 0$, and $s \xrightarrow{e} s''$ is a discrete transition and $s'' \xrightarrow{\delta} s'$ is a timed transition.

Lemma 2. *Let A be a TA, $c \geq c_{max}(A)$ and s_1, s_2 be two c-equivalent states of A. Then, $s_1 + \delta$ and $s_2 + \delta$ are c-equivalent for any $\delta \in \mathsf{R}$. Moreover, for any edge e, if $s_1 \xrightarrow{e} s_1'$ is a discrete transition, then $s_2 \xrightarrow{e} s_2'$ is a discrete transition such that s_1' and s_2' are c-equivalent.*

Intuitively, the above lemma says that two states of A have essentially the same executions, if they agree on their clock values, except perhaps for those clocks which have grown greater than $c_{max}(A)$ [2]. The result will be used in section 3.1 for proving correctness of the reachability algorithm.

Runs. A *run* of A starting from state s_1 and reaching state s_k is a sequence $\rho = s_1 \xrightarrow{e_1} s_1' \xrightarrow{\delta_1} s_2 \xrightarrow{e_2} \cdots \xrightarrow{\delta_k} s_k$, such that $s_1 = s$ and for all $i = 1, ..., k$, s_i' is the e_i-successor of s_i and s_{i+1} is the δ_i-successor of s_i'. We say that s_k is reachable from s_1.

Reachability problem and diagnostic runs. The *reachability problem* for A is, given a set of initial states S_1 and a set of target states S, find whether there exists a run starting from some state $s_1 \in S_1$ and reaching some state $s \in S$. To provide timed diagnostics means to exhibit such a run.

[2] In fact, c-equivalence is a *bisimulation*.

3 Reachability with diagnostics

Consider a TA A, and two sets S_1 and S, of initial and target states, respectively. We check whether S_1 can reach S and, if so, provide timed diagnostics. Our method of consists in two steps:

- First, we apply the reachability algorithm of [DT98] to check whether there exists some run from S_1 to S. If reachability succeeds, the algorithm generates an abstract path, that is, a sequence $\pi = S_1 \xrightarrow{e_1} \cdots \xrightarrow{e_k} S$, where S_i are sets of states, and e_i are edges of A, for $i = 1, ..., k$.
- Then, we extract a run $\rho = s_1 \xrightarrow{e_1}\xrightarrow{\delta_1} s_2 \cdots \xrightarrow{e_k}\xrightarrow{\delta_k} s$, such that ρ is *inscribed* in π, that is: (1) $s_1 \in S_1$, (2) $s - \delta_k, s \in S$, and (3) for each $1 \leq i < k$, $s_{i+1} - \delta_i, s_{i+1} \in S_{i+1}$.

3.1 Checking reachability

A *zone* of A is a set of states $\{(q, \mathbf{v}) \mid \mathbf{v} \in \zeta\}$, where ζ is an $\mathcal{X}$-polyhedron. For simplicity, we denote such a zone by (q, ζ).

Given a zone (q, ζ), an edge $e = (q, \zeta', X, q')$ and a natural constant c, we define

$$\mathsf{post}(q, \zeta, e, c) \stackrel{\text{def}}{=} \Big(q', \mathsf{close}(\nearrow((\zeta \cap \zeta')[X := 0]), c)\Big)$$

Notice that the result of $\mathsf{post}()$ is a zone, since polyhedra are closed with respect to the operations of intersection, clock reset, projection and c-closure. Also observe that the operator is *monotonic*, that is, $\zeta_1 \subseteq \zeta_2$ implies $\mathsf{post}(q, \zeta_1, e, c) \subseteq \mathsf{post}(q, \zeta_2, e, c)$.

The essential properties of $\mathsf{post}()$ are stated in the following lemma.

Lemma 3. *If $(q', \zeta') = \mathsf{post}(q, \zeta, e, c)$, then:*

1. *For each $\mathbf{v} \in \zeta$ and each $\delta \in \mathsf{R}$, if $(q, \mathbf{v}) \xrightarrow{e}\xrightarrow{\delta} (q', \mathbf{v}')$, then $\mathbf{v}' \in \zeta'$.*
2. *For each $\mathbf{v}' \in \zeta'$, there exist $\delta \in \mathsf{R}$, $\mathbf{v} \in \zeta$ and $\mathbf{v}'' \in \zeta'$, such that $(q, \mathbf{v}) \xrightarrow{e}\xrightarrow{\delta} (q', \mathbf{v}'')$ and $\mathbf{v}', \mathbf{v}''$ are c-equivalent.*

Intuitively, $\mathsf{post}(q, \zeta, e, c)$ contains all successor states of (q, ζ), by a discrete e-transition and then a timed transition. Since the final result is closed under c-equivalence, some states might be added which are not direct successors, however, they are c-equivalent to some direct successor.

Based on the above lemma, we develop the algorithm of figure 2. The algorithm uses a depth-first search (DFS) to explore all successor zones of the initial zone (q_1, ζ_1). The search stops when either a zone is found which intersects the target zone (q, ζ) (line 1) or no more zones are left to be explored.

Visit is the set of zones already visited, initially empty. Each new successor zone is inserted in *Visit* when the `DFS` procedure is called recursively (line 2).

For each out-going edge of the current zone (q_1, ζ_1), `DFS` generates its successor (line 3). Empty successors are ignored (line 4). The same is true with any

```
/* Precondition: c ≥ max(c_max(A), c_max(ζ)) */
  DFS ((q1, ζ1), (q, ζ)) {
    if (q1 = q ∧ ζ1 ∩ ζ ≠ ∅) then return "Yes" ;                    (1)
    Visit := Visit ∪ {(q1, ζ1)} ;                                   (2)
    for each (e ∈ out(q1)) do
      (q', ζ') := post(q1, ζ1, e, c) ;                              (3)
      if (ζ' = ∅) then continue ;                                   (4)
      else if (∃(q', ζ'') ∈ Visit . ζ' ⊆ ζ'') then continue ;       (5)
      else DFS ((q', ζ'), (q, ζ)) ;
    end for each
  }
```

Fig. 2. A DFS for Yes/No reachability.

successor (q', ζ') which is contained in an already visited zone (q', ζ'') (line 5): since $\mathsf{post}()$ is monotonic, all successors of (q', ζ') are contained in (q', ζ''), thus, (q', ζ') does not have to be further explored.

The algorithm generates pairs (q, ζ), where q is a discrete state and ζ is a c-closed polyhedron. By definition there is a finite number of discrete states and by lemma 1 there is a finite number of c-closed polyhedra, thus, the algorithm terminates. Correctness follows from lemmas 2 and 3.

As presented, the algorithm of figure 2 only gives a yes/no answer to the reachability problem. It is easy to see how the abstract path reaching (q, ζ) can also be returned: a DFS is usually implemented using a stack to keep the current sequence of zones and edges explored. This sequence corresponds exactly to the abstract path. In what follows, we show how to obtain more, namely, how to extract a run from the abstract path.

3.2 Extracting runs from abstract paths

Let $\pi = (q_1, \zeta_1) \stackrel{e_1}{\rightarrow} \cdots \stackrel{e_l}{\rightarrow} (q_{l+1}, \zeta_{l+1})$ be the abstract path returned by the DFS of figure 2, where for each $i = 1, ..., l$, $(q_{i+1}, \zeta_{i+1}) = \mathsf{post}(q_i, \zeta_i, e_i, c)$ and $S_{l+1} = (q, \zeta_{l+1})$, with $\zeta_{l+1} \cap \zeta \neq \emptyset$. For simplicity, we assume that $\zeta_{l+1} \subseteq \zeta$ (otherwise, we can just replace ζ_{l+1} by $\zeta_{l+1} \cap \zeta$).

We show how to build a run inscribed in π. The run is built in two passes, first backwards and then forwards:

- Backward pass: initially we choose $s_{l+1} \in (q_{l+1}, \zeta_{l+1})$ and then successively compute $\delta_i \in \mathsf{R}$, $s_i \in (q_i, \zeta_i)$, for $i = l, ..., 1$, such that $s_i \stackrel{e_i}{\rightarrow}\stackrel{\delta_i}{\rightarrow} s'_{i+1}$, for some s'_{i+1} which is c-equivalent to s_{i+1}.
- Forward pass: starting from $s_1 \in (q_1, \zeta_1)$, we compute s'_i, for $i = 2, ..., l+1$, based on δ_i, e_i. The final run is $s_1 \stackrel{e_1}{\rightarrow}\stackrel{\delta_1}{\rightarrow} s'_2 \cdots \stackrel{e_l}{\rightarrow}\stackrel{\delta_l}{\rightarrow} s'_{l+1}$.

Intuitively, the backward pass generates an invalid run which might contain some "jumps" among c-equivalent states. The forward pass corrects the run by "adjusting" the clocks which have grown greater than $c_{max}(A)$.

Before describing the two passes in detail, we show how choosing a state in a zone (q, ζ) can be done effectively. In fact, this comes down to extracting a valuation $\mathbf{v} \in \zeta$. In the sequel, we assume that the set of clocks is $\mathcal{X} = \{x_1, ..., x_n\}$.

Extracting valuations from polyhedra. An *k-incomplete valuation* is a valuation $\mathbf{v}$ on $\{x_1, ..., x_k\}$. We say that $\mathbf{v}$ can be *completed* in ζ if there exists an $\mathcal{X}$-valuation $\mathbf{v}' \in \zeta$, such that $\mathbf{v}'(x_j) = \mathbf{v}(x_j)$, for all $j \leq k$. Completing $\mathbf{v}$ in ζ means finding such a $\mathbf{v}'$. Notice that we permit $k = 0$, so that completing a 0-incomplete valuation in ζ means extracting a valuation from ζ.

Lemma 4. *Let ζ be an $\mathcal{X}$-polyhedron and $\mathbf{v}$ be an k-incomplete valuation. It takes $O(n^2)$ time to complete $\mathbf{v}$ in ζ, or find that this is not possible.*

Proof. Let $\zeta = \bigwedge_{0 \leq i \neq j \leq n} x_i - x_j \prec_{i,j} c_{i,j}$. For $i = 0, ..., k$, we define:

$$\delta_i = \begin{cases} 0, & \text{if } i = 0 \\ \mathbf{v}(x_i), & \text{if } 1 \leq i \leq k \end{cases}$$

Then, for $i = k+1, ..., n$, we choose δ_i such that:

$$\forall 0 \leq j < i \ \ . \ -c_{j,i} \prec_{j,i} \delta_i \prec_{i,j} c_{i,j}$$

If such a δ_i cannot be chosen for some i, then $\mathbf{v}$ cannot be completed. Otherwise, we let $\mathbf{v}'(x_i) = \delta_i$, for $i = 1, ..., n$. It is easy to see that $\mathbf{v}' \in \zeta$.

Regarding complexity, in the worst case we have $i = 0$, meaning that we have to perform $n \cdot (n-1) + n$ comparisons and additions of bounds.

Backward pass. It suffices to show how the computation is done for a single step, say, $(q_1, \zeta_1) \xrightarrow{e} (q_2, \zeta_2)$. That is, given $\mathbf{v}_2 \in \zeta_2$, we shall show how to compute $\delta \in \mathsf{R}$ and $\mathbf{v}_1 \in \zeta_1$ such that $(q_1, \mathbf{v}_1) \xrightarrow{e}\xrightarrow{\delta} (q_2, \mathbf{v}_2')$, and $\mathbf{v}_2, \mathbf{v}_2'$ are c-equivalent.

Finding δ can be done by "pulling $\mathbf{v}_2$ backward in time", until some clock reset in e reaches 0. More precisely, if $\mathsf{reset}(e) = \emptyset$ then we let $\delta = 0$, otherwise we let $\delta = \mathbf{v}_2(x)$, for some $x \in \mathsf{reset}(e)$.

Now, let $\mathbf{v}_3 = \mathbf{v}_2 - \delta$. By definition, we have $\mathbf{v}_3 \in \zeta_2$ and $(q_2, \mathbf{v}_3) \xrightarrow{\delta} (q_2, \mathbf{v}_2)$. It remains to find $\mathbf{v}_1 \in \zeta_1$ such that $(q_1, \mathbf{v}_1) \xrightarrow{e} (q_2, \mathbf{v}_4)$ and $\mathbf{v}_4$ and $\mathbf{v}_3$ are c-equivalent, which implies that $\mathbf{v}_4 + \delta$ and $\mathbf{v}_2$ are also c-equivalent.

Without loss of generality, we assume that there exists $0 \leq k \leq n$ such that the clocks $x_1, ..., x_k$ are not reset in e and for each $j = 1, ..., k$, $\mathbf{v}_2(x_j) \leq c$.

First, $\mathbf{v}_1$ should satisfy $\mathsf{guard}(e)$. Moreover, since clocks $x_1, ..., x_k$ are not reset in e, they should have the same value in $\mathbf{v}_1$ and $\mathbf{v}_3$. Then, we let $\mathbf{v}$ be a k-incomplete valuation, such that $\mathbf{v}(x_i) = \mathbf{v}_3(x_i)$, for $i = 1, ..., k$. Using lemma 4, we can complete $\mathbf{v}$ in $\zeta_1 \cap \mathsf{guard}(e)$. This is always possible, by the second part of lemma 3.

Therefore, we define $\mathbf{v}_1$ to be the completed valuation. If we let $\mathbf{v}_4 = \mathbf{v}_1[\mathsf{reset}(e) := 0]$, we have:

- for $i = 1, ..., k$, $\mathbf{v}_4(x_i) = \mathbf{v}_3(x_i)$;
- for $i = k+1, ..., n$,
  - if $x_i \in \mathsf{reset}(e)$, then $\mathbf{v}_4(x_i) = \mathbf{v}_3(x_i) = 0$,
  - otherwise, $\mathbf{v}_4(x_i) > c$ and $\mathbf{v}_3(x_i) > c$.

That is, $\mathbf{v}_4$ and $\mathbf{v}_3$ are c-equivalent.

Regarding the complexity of the backward pass, observe that for each step, it takes $O(n)$ time to find the delay δ and $O(n^2)$ time to complete the valuation [3]. Therefore, the whole pass can be performed in time $O(l \cdot n^2)$.

Forward pass. This pass is easy. We start from $s_1 = (q_1, \mathbf{v}_1)$, as computed in the backward pass. Then, for $i = 1, ..., l+1$, we compute $\mathbf{v}'_i$ by "adjusting" $\mathbf{v}_i$ as follows.

- $\mathbf{v}'_1 = \mathbf{v}_1$;
- for $i = 2, ..., l+1$, $\mathbf{v}'_i = (\mathbf{v}'_{i-1}[\mathsf{reset}(e_i) := 0]) + \delta_i$.

Using lemma 2 and induction on l, it is easy to prove that the resulting run is valid, that is, $(q_i, \mathbf{v}'_i) \xrightarrow{e_i} \xrightarrow{\delta_i} (q_{i+1}, \mathbf{v}'_{i+1})$, for all $i = 1, ..., l$.

The complexity of the forward pass is $O(l \cdot n)$. Therefore, the complexity of computing the whole run is $O(l \cdot n^2)$.

Example. Consider the simple TA shown in figure 3. We are interested in reachability of the target zone (q_3, true) from the initial zone $(q_1, x = y)$. Let e_1 be the edge from q_1 to q_2 and e_2 the edge from q_2 to q_3. The algorithm of figure 2 succeeds, returning the abstract path $(q_1, x = y) \xrightarrow{e_1} (q_2, y = x+2) \xrightarrow{e_2} (q_3, y > x+2)$. Notice that for this example $c = 2$ and before applying $\mathsf{close}()$, the polyhedron associated to q_3 is $y = x + 4$.

For the backward pass, we start by choosing $\mathbf{v}_3 \in y > x + 2$, say, $\mathbf{v}_3 = (x = 0, y = 3)$. This gives $\delta_3 = 0$. Then, we must complete a 0-incomplete valuation in $y = x + 2 \wedge x = 2$, which gives us $\mathbf{v}_2 = (x = 2, y = 4)$. Since x is reset in e_1, we get $\delta_2 = 2$. Finally, we have to complete a 0-incomplete valuation in $y = x \wedge x = 2$, which gives us $\mathbf{v}_1 = (x = 2, y = 2)$. At the end of the backward pass, we have the sequence $(q_1, x = 2, y = 2) \xrightarrow{e_1} (q_2, x = 0, y = 2) \xrightarrow{2} (q_2, x = 2, y = 4) \xrightarrow{e_2} (q_3, x = 0, y = 3)$. This is not a valid run, since there is a "jump" of clock y on the e_2-transition.

The forward pass adjusts $\mathbf{v}_3$ to $\mathbf{v}'_3 = (x = 0, y = 4)$, yielding the final (valid) run: $(q_1, x = 2, y = 2) \xrightarrow{e_1} (q_2, x = 0, y = 2) \xrightarrow{2} (q_2, x = 2, y = 4) \xrightarrow{e_2} (q_3, x = 0, y = 4) \xrightarrow{0} (q_3, x = 0, y = 4)$.

[3] Completing a valuation in the intersection of more than one polyhedra, say, $\zeta_1 \cap \cdots \cap \zeta_m$, multiplies the complexity of the operation by only a constant factor m.

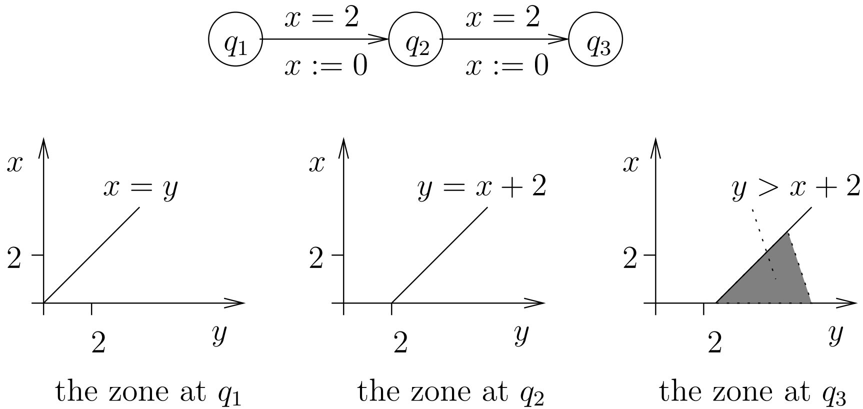

Fig. 3. A simple example.

4 Case study: Bang&Olufsen's Collision-Detection Protocol

We have implemented the technique presented in the previous section in the real-time verification tool Kronos, which can now provide timed diagnostics when reachability analysis succeeds [4].

We have used Kronos to verify the industrial Bang&Olufsen protocol, treated with Uppaal in [HSLL97]. The TA models of Kronos and Uppaal are essentially the same, so that translating the specification of [HSLL97] to Kronos format was almost straightforward. The protocol is only briefly presented here; the reader is referred to the above paper for more details.

Brief description and modeling. The role of the protocol is to ensure collision detection in a distributed environment of components exchanging messages through a common multiple-access bus. The system modeled has two transmission components A and B (identical up to renaming) and the bus. Since we are interested only in the collision-detection protocol, the reception components are not modeled. A and B consist each of 3 sub-components, namely, the sender, the detector and the frame generator. The sender handles transmission of messages, which are grouped in *frames*. The latter are generated by the frame generator. The detector is responsible for collision detection.

The components along with their communication channels are shown in figure 4. Each component is modeled as an automaton. The two senders are modeled

[4] The implementation is actually compatible with the new features of Kronos, including discrete variables (of type boolean, bounded integer or enumerative), message passing, and a variable-dimension DBM library which exploits the *activity* of clocks [DT98] to reduce the size of the state space.

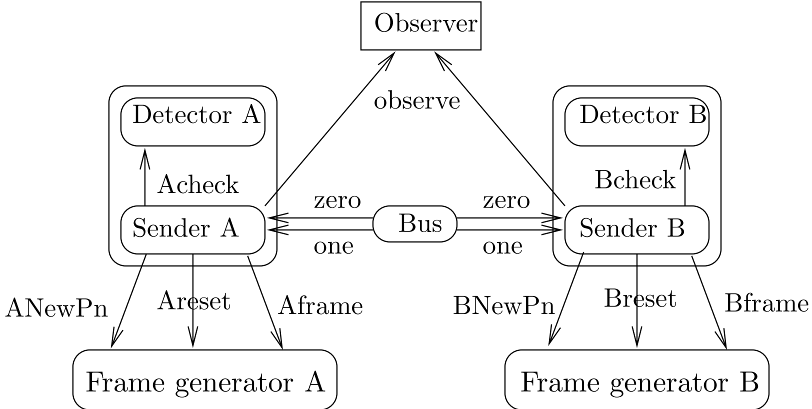

Fig. 4. BANG&OLUFSEN's protocol: general architecture.

by timed automata whereas the rest of the automata are untimed [5]. Figure 5 shows the TA for sender A. The figure is merely intended to give an impression of the complexity of the case study and the modeling issues involved. In particular, UPPAAL uses so-called "committed locations", which are not a standard feature of KRONOS. However, they can be easily modeled as described in appendix A.

The most interesting feature of the protocol is its timing constraints, which concern the frequency of senders' polling on the bus, the encoding of messages and the waiting delay required before retransmitting after a collision. For instance, a sender samples the value of the bus (1 for high voltage, 0 for low voltage) twice every 781 micro-seconds. Also, there are 5 different types of messages and the i-th message is encoded by the presence of a 1 on the bus, for $2 \cdot 1562 \cdot i$ micro-seconds. Finally, the *jamming signal*, after a collision, is a continuous 1 on the bus for 25 milli-seconds [6].

Verification. The protocol must ensure collision detection, that is, if a frame sent by a sender is destroyed by the other sender (collision), then both senders shall detect this. According to [HSLL97], collision happens when the boolean expression

$$\phi_{col} \stackrel{\text{def}}{=} \neg(A_Pf \Leftrightarrow A_S_1 \wedge A_Pn \Leftrightarrow A_S_2)$$

[5] The observer automaton shown in the figure is not part of the system itself, but is added to monitor the system for possible errors, as we explain below.

[6] Notice that duration constants vary from 40 micro-seconds to 0.5 seconds and have no common divisor (look at figure 5). This implies a very small time quantum, namely, one micro-sec, which results in very large constants in guards and invariants. Consequently, enumerative approaches based on discretization are not well-suited for this case study, since time units have to be counted one-by-one, leading to state explosion.

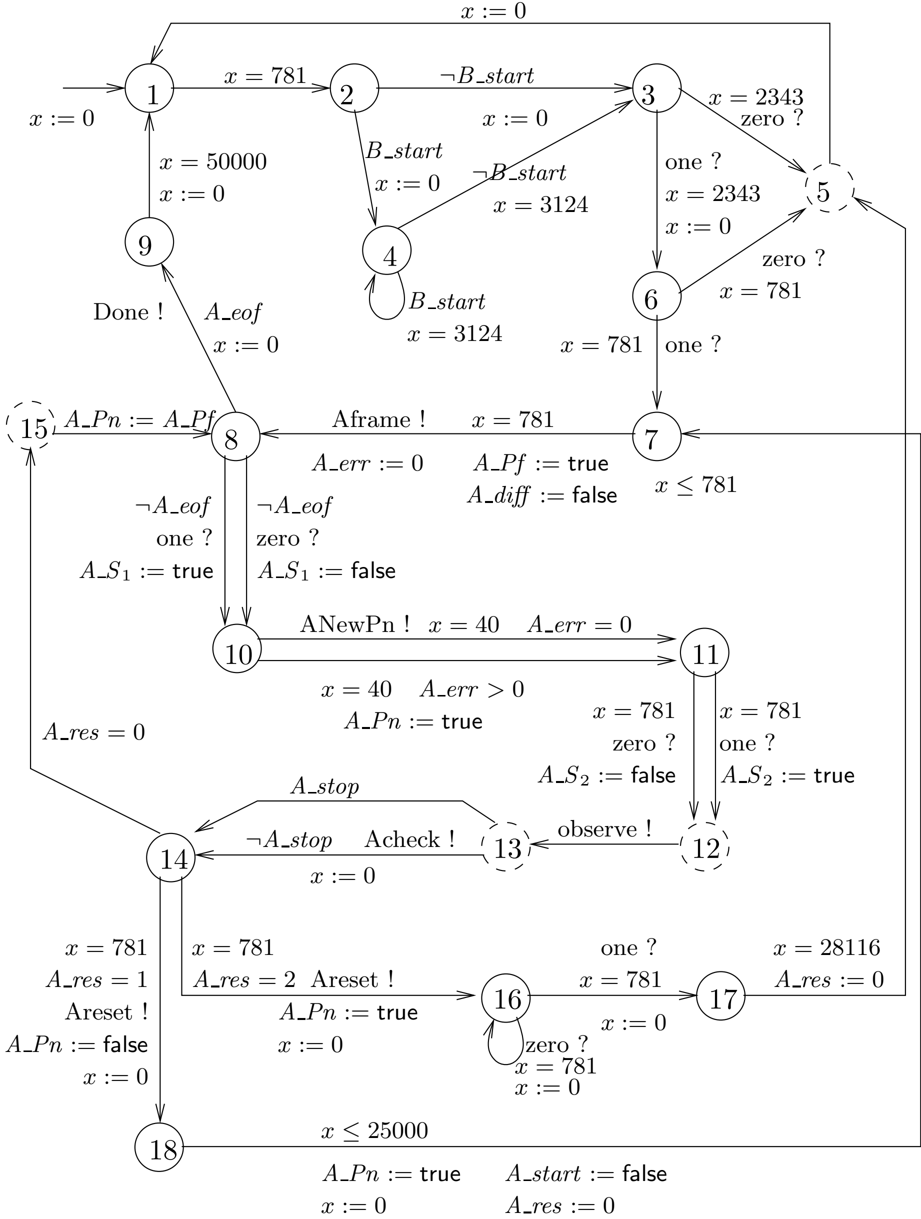

Fig. 5. BANG&OLUFSEN's example: the TA for sender A.

evaluates to false at the moment A_S_2 is assigned (transition from control state 11 to 12 in figure 5). A collision is detected when the result of the detector automaton (called by signal "Acheck !") is $A_res = 1$ or $A_res = 2$, whereupon the sender emits an "Areset !" signal.

Now we can model the requirement in terms or reachability of the "error" state of the observer automaton shown in figure 6. The observer starts at its left-most state and moves to its middle state when a collision happens. If the collision is detected before the the sender finishes transmitting (modeled by signal "Done !") then the observer returns to its initial state, otherwise it goes to the error state.

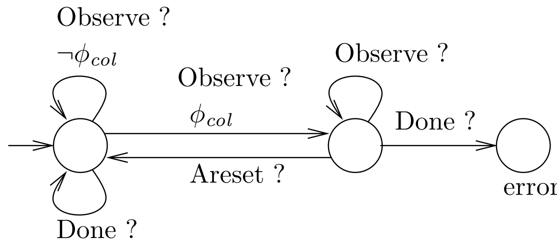

Fig. 6. Bang&Olufsen's example: the observer automaton.

Results and performance. [HSLL97] present two versions of the protocol: the initial version contains an error (Uppaal provides an abstract counter-example); then, the frame-generator automaton is slightly modified and the authors of [HSLL97] claim this version to be correct. However, we have found a counter-example in both versions. The counter-example of the "corrected" version is generated by Kronos in 25 seconds on a Sparc 20 [7].

The complete diagnostic run is 1951 discrete/timed steps long. Here, we show only its head and its tail, as produced by the tool [8]:

```
<> - 0 - <>                              -- y:=0 "b_go" -->
<y:0> - 40 - <y:40>                      -- x:=0 "a_go" -->
<x:0, y:40> - 741 - <x:741, y:781>       -- y:=0 "b_start_frame" -->
<x:741, y:781> - 40 - <x:781, y:821>     -- "a_silent" -->
<x:781, y:40> - 0 - <x:781, y:40>        -- x:=0 "a_start_frame" -->
<x:781, y:40> - 2303 - <x:3084, y:2343>  -- "b_silent" -->
```

[7] As in [HSLL97], in order to obtain a fast answer, we have used a simplified model where not the whole variety of messages could be generated.

[8] There are too many discrete variables, thus, only the clock valuation is shown for each state. Clocks x and y correspond to senders A and B, respectively. The initial valuation is trivial since no clocks are initially active. In the second valuation, only y is active.

```
<x:2303, y:2343> - 0 - <x:2303, y:2343>  -- y:=0 "b_one" -->
<x:2303, y:2343> - 40 - <x:2343, y:2383> -- x:=0 "a_one" -->
<x:2343, y:40> - 741 - <x:3084, y:781> -- y:=0 "b_one" -->
<x:741, y:781> - 40 - <x:781, y:821>   -- x:=0 "a_one" -->
<x:781, y:40> - 741 - <x:1522, y:781>  -- y:=0 "b_frame" -->
...
<x:741, y:781> - 40 - <x:781, y:821>  -- "b_observe_ok" -->
<x:781, y:40> - 0 - <x:781, y:40>     -- "b_stopped" -->
<x:781, y:40> - 0 - <x:781, y:40>     -- x:=0 "a_zero" -->
<x:781, y:40> - 741 - <x:1522, y:781> -- "a_diff_pf_s1" -->
<x:741, y:781> - 0 - <x:741, y:781>   -- "a_stopped" -->
<x:741, y:781> - 0 - <x:741, y:781>   -- y:=0 "b_nocol" -->
<x:741, y:781> - 40 - <x:781, y:821>  -- "b_pf0" -->
<x:781, y:40> - 0 - <x:781, y:40>     -- "b_zero" -->
<x:781, y:40> - 0 - <x:781, y:40>     -- "b_new_pn" -->
<x:781, y:40> - 0 - <x:781, y:40>     -- x:=0 "a_nocol" -->
<x:781, y:40> - 40 - <x:821, y:80>    -- "a_pf0" -->
<x:40, y:80> - 0 - <x:40, y:80>       -- "a_zero" -->
<x:40, y:80> - 0 - <x:40, y:80>       -- "a_new_pn" -->
```

Intuitively, the error seems to be due to the following reasons:

1. The two senders start transmitting with a difference of *exactly* 40 μ-seconds. Due to this fact and the way the sampling of the bus is performed, collision remains undetected until the last message of the frame is sent.
2. In the last message of the frame (a message signaling *end-of-frame*) the collision detection procedure is disarmed. This can be seen in the tail of the diagnostic run above: instead of the action `a_check` calling the collision detection procedure, we see the action `a_stopped`, which means that boolean variable *A_stop* is set. Therefore, collision is not detected by A. The situation is the same for sender B.

5 Conclusions

We have shown how to compute exact diagnostics for timed systems with respect to reachability properties. Our technique has enhanced the verification tool Kronos with a useful feature, which makes debugging of the model and discovery of real system flaws much easier.

Related work. Timed diagnostics have been considered independently in [LPY95]. However, only the existence of a run inscribed in a symbolic path is stated and no method is given on how to actually extract the run. Moreover, the symbolic reachability of [LPY95] does not contain the c-closure operation. This makes the extraction of runs simpler, but without c-closure termination is not generally ensured.

Recently, [AKV98] have developed an algorithm which, given a sequence of edges, produces a corresponding run, if one exists. This algorithm has complexity $O(l \cdot n^2)$ as ours, and can also be used to extract a timed diagnostic from a symbolic path.

Acknowledgments. This work would not have been possible without the help of Marius Bozga for extending KRONOS without discrete variables. I would like also to thank him for his help in understanding the counter-example trace in BANG&OLUFSENs protocol.

References

ACD93. R. Alur, C. Courcoubetis, and D.L. Dill. Model checking in dense real time. *Information and Computation*, 104(1):2–34, 1993.

AKV98. R. Alur, R.P. Kurshan, and M. Viswanathan. Membership questions for timed and hybrid automata. In *RTSS'98*, 1998.

BTY97. A. Bouajjani, S. Tripakis, and S. Yovine. On-the-fly symbolic model checking for real-time systems. In *Proc. of the 18th IEEE Real-Time Systems Symposium, San Francisco, CA*, pages 232–243. IEEE, December 1997.

Dil89. D.L. Dill. Timing assumptions and verification of finite-state concurrent systems. In J. Sifakis, editor, *Automatic Verification Methods for Finite State Systems*, Lecture Notes in Computer Science 407, pages 197–212. Springer–Verlag, 1989.

DOTY96. C. Daws, A. Olivero, S. Tripakis, and S. Yovine. The tool KRONOS. In *Hybrid Systems III, Verification and Control*, volume 1066 of *LNCS*, pages 208–219. Springer-Verlag, 1996.

DT98. C. Daws and S. Tripakis. Model checking of real-time reachability properties using abstractions. In *Tools and Algorithms for the Construction and Analysis of Systems '98, Lisbon, Portugal*, volume 1384 of *LNCS*. Springer-Verlag, 1998.

HNSY94. T.A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic model checking for real-time systems. *Information and Computation*, 111(2):193–244, 1994.

HSLL97. K. Havelund, A. Skou, K. Larsen, and K. Lund. Formal modelling and analysis of an audio/video protocol: An industrial case study using Uppaal. In *Proceedings of the 18th IEEE Real-Time Systems Symposium San Francisco, CA*, pages 2–13, December 1997.

LPY95. K. Larsen, P. Pettersson, and W. Yi. Diagnostic model-checking for real-time systems. In *4th DIMACS Workshop on Verification and Control of Hybrid Systems*, 1995.

Yov93. S. Yovine. *Méthodes et outils pour la vérification symbolique de systèmes temporisés.* PhD thesis, Institut National Polytechnique de Grenoble, 1993. In french.

A Modeling "committed locations"

Committed locations are simply discrete states modeling *atomic* execution. Informally, the semantics are as follows, for a network of TA. When an automaton

A enters a committed location, it has to exit immediately, and no other automaton takes a discrete step meanwhile.

To model committed locations, we introduce a global boolean variable *atom* and a global clock z (a single boolean variable and a single clock suffice, no matter how many the committed locations are). The invariant that must hold during execution is that *atom* is set iff some automaton is in a committed location and that the time spend in committed locations is zero.

For each TA A in the global system, if $e = (q, \zeta, X, q')$ is an edge of A, then:

- If q is not committed, then we add the boolean guard $\neg atom$ to e.
- If q is committed, then we add the clock guard $z = 0$ to e.
- If q' is committed, then we add the assignment $atom := \mathsf{true}$ and the clock reset $z := 0$ to e.
- If q' is not committed, then we add the assignment $atom := \mathsf{false}$ to e.

The construction is illustrated in figure 7.

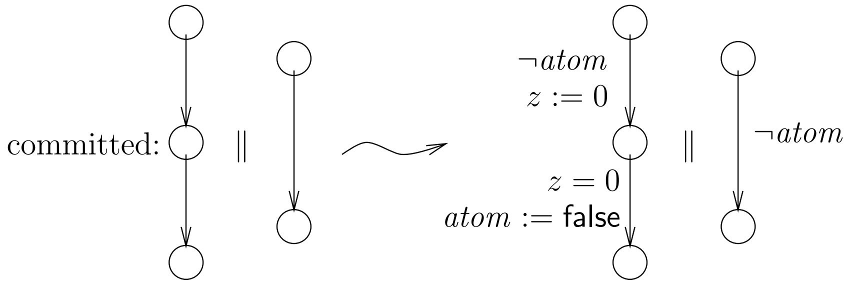

Fig. 7. Modeling committed locations with an auxiliary boolean variable and clock.

134-502-71

10.1007/b107031130005