Efficient Reachability Analysis and Refinement Checking of Timed Automata Using BDDs

Dirk Beyer

Software Systems Engineering Research Group
Technical University Cottbus, Germany
db@informatik.tu-cottbus.de

Keywords. Formal verification, Real-time systems, Timed Automata

1 Introduction

For the formal specification and verification of real-time systems we use the modular formalism Cottbus Timed Automata (CTA), which is an extension of timed automata [AD94]. Matrix-based algorithms for the reachability analysis of timed automata are implemented in tools like Kronos, Uppaal, HyTech and Rabbit. A new BDD-based version of Rabbit, which supports also refinement checking, is now available.

For the representation of the models we use an *integer semantics* for *closed timed automata*. Using this discretization, we are able to use a unique representation of the discrete state space (given by the locations) and the continuous state space (given by the clocks). We use an estimate-based strategy for variable ordering which dramatically compresses the BDD representation of the transition relation and the reachable configurations and thus leads to much more efficient verification.

The restricted applicability of reachability analysis due to the high time complexity of the analysis for large models leads to the need of refinement checking for verification. We implemented an algorithm for checking the existence of a simulation relation to investigate the opportunities of refinement checking for Cottbus Timed Automata.

Section 2 introduces our notation for modular modeling of real-time systems: we recall the formal definition of timed automata and our integer semantics for closed timed automata. In Sect. 3 we describe our implementation of reachability analysis and, in more detail, in Sect. 4 we define the corresponding refinement checking. In Sects. 3 and 4, we present performance results for some example models.

2 Cottbus Timed Automata

We start with an informal definition of Cottbus Timed Automata (CTA), which is a modeling concept providing means for modular design [BR98]. A formal definition and the complete semantics of CTA are given in [BR01]. A CTA system description consists of a set of modules. One of them is the top module, which models the whole system. The other modules are used as templates. They can be instantiated several times in different modules. Thus, it is possible to express a hierarchical structure of the system, and to define replicated components of a system just once.

T. Margaria and T. Melham (Eds.): CHARME 2001, LNCS 2144, pp. 86–91, 2001.

Each module is named by an identifier. **(1)** The **interface** contains the declarations of *clock variables* and *synchronization labels*, each of them has a *restriction type* to control the access to the component. We distinguish the restriction types INPUT, OUTPUT, MULTIPLY RESTRICTED, and LOCAL. **(2)** A module contains a **timed automaton** as defined below. **(3)** The **initial condition** is a predicate over the module's variables and locations. **(4)** A module may contain **instances** of previously defined modules.

We now define closed timed automata and their integer semantics. *Clock constraints* are allowed as invariants and guards of a timed automaton. Let X be a set of clocks. Atomic clock constraints over X are comparisons of a clock with a time constant from $\mathbb{N}$, the set of natural numbers (including 0). Clock constraints are conjunctions of atomic clock constraints. Formally, the set $\Phi(X)$ of clock constraints over X for closed timed automata is generated by $\varphi := x \leq c \mid x \geq c \mid \varphi \wedge \varphi$, with $x \in X$ and $c \in \mathbb{N}$. For closed timed automata it is sufficient to use only integer clock values for the computation of reachable locations.

The *clock assignments* $Val_I(X)$ of X are the total functions from X into the set of natural numbers $\mathbb{N}$. For a clock constraint $\varphi \in \Phi(X)$, $[\![\varphi]\!]$ denotes the set of all clock assignments of X that satisfy φ. The clock assignment which assigns the value 0 to all clocks is denoted by v^0. For a timed automaton $\mathcal{A}$ with a clock x, $C_{\mathcal{A}}(x)$ denotes the greatest constant to which x is compared within a clock constraint of $\mathcal{A}$. For $v \in Val_I(X)$ and $\delta \in \mathbb{N}$, $v \oplus \delta$ is the clock assignment of X that assigns the value $min\,(v(x) + \delta, C_{\mathcal{A}}(x) + 1)$ to each clock x. For $v \in Val_I(X)$ and $Y \subseteq X$, $v[Y := 0]$ denotes the clock assignment of X that assigns the value 0 to each clock in Y and leaves the other clocks as in v.

A **closed timed automaton** $\mathcal{A}$ is a tuple $(L, L^0, X, \Sigma, I, E)$, where L is a finite set of *locations*, $L^0 \subseteq L$ is a set of *initial locations*, X is a finite set of *clocks*, Σ with $\Sigma \cap \mathbb{N} = \emptyset$ is a finite set of *synchronization labels*, I is a total function that assigns an *invariant* from $\Phi(X)$ to each location in L, $E \subseteq L \times \Sigma \times \Phi(X) \times 2^X \times L$ is a set of *switches*. A switch (l, a, φ, Y, m) represents a transition labeled with synchronization label a from location l to location m. The guard φ has to be satisfied to enable the switch. The switch resets all clocks in Y to the value 0.

The semantics of a timed automaton is defined by associating a transition system with it. Let $\mathcal{A} = (L, L^0, X, \Sigma, I, E)$ be a closed timed automaton. The **integer semantics** $[\![\mathcal{A}]\!]_I$ of $\mathcal{A}$ is the transition system $(L \times Val_I(X), L^0 \times \{v^0\}, \Sigma \cup \mathbb{N}, \rightarrow_I)$ with the following timed and discrete transitions:

- For $(l, v), (m, w) \in L \times Val_I(X)$ and $\delta \in \mathbb{N}$, $(l, v) \xrightarrow{\delta}_I (m, w)$ holds iff $l = m$, $w = v \oplus \delta$, $v \in [\![I(l)]\!]$ and $w \in [\![I(l)]\!]$.
- For $(l, v), (m, w) \in L \times Val_I(X)$ and $a \in \Sigma$, $(l, v) \xrightarrow{a}_I (m, w)$ holds iff there exists an $(l, a, \varphi, Y, m) \in E$ with $v \in [\![\varphi]\!]$ and $w = v[Y := 0]$.

In the following we define runs and reachable configurations for a transition system $\mathcal{T} = (S, S^0, \Sigma_{\mathcal{T}}, \rightarrow)$. Let $(s_0, s_1, ..., s_k)$ be a finite sequence of configurations, $a_0, a_1, ..., a_{k-1} \in \Sigma_{\mathcal{T}}$, $s_0 \in S^0$, and $s_i \xrightarrow{a_i} s_{i+1}$ for all $i \in \{0, 1, ..., k-1\}$. Then $(s_0, s_1, ..., s_k)$ is a **run** of $\mathcal{T}$. $Run(\mathcal{T})$ denotes the set of runs of $\mathcal{T}$. The configuration s_k is **reachable**. $Reach(\mathcal{T})$ denotes the set of reachable configurations (shorter: reachable set) of $\mathcal{T}$.

In our tool implementation we use the integer semantics for closed timed automata as defined above. This integer semantics is equivalent to the usual, continuous one regarding the set of reachable locations. More details and a formal proof are given in [Bey01]. To verify safety properties of a CTA model we provide two techniques: reachability analysis and refinement checking as described in the following two sections.

3 Reachability Analysis

For verification of a safety property, i.e. whether a configuration of a special set of invalid configurations is reachable or not, we compute at first the set of reachable configurations. Secondly, we intersect the reachable set with the set of invalid configurations. If the intersection is empty, then the safety property is fulfilled. Variants of this algorithm are possible for speed-up, e.g. on-the-fly analysis. We extended our existing matrix-based model checker **Rabbit** by the BDD-based reachability analysis of closed timed automata. Experience with finite automata shows that the efficiency critically depends on the choice of several parameters. In this section we sketch how our implementation determines these parameters.

Variable Ordering. We take the pre-order linearization of the CTA model as initial variable ordering. This implies that we consider the modeler's decision to encapsulate some components together within one module, i.e. local components of a module are assigned to neighboring positions within the variable ordering. Then we apply another heuristic to optimize the ordering respecting a size estimate for the BDD of the set of reachable configurations, which is derived from our upper bound for the transition relation as described and proven in [Bey01].

Partial Transition Relations. Usually the transition relation $\rightarrow$ is represented as implicit union of a timed transition relation $\xrightarrow{1}$ and discrete transition relations $\xrightarrow{a}$ for each synchronization label a. Experiments have shown that applying such partial transition relations sequentially is more efficient than using the union of these relations as monolithic transition relation.

Order of Transitions. Using several partial transition relations, we have to determine the order of their application. The intermediate sets of reached configurations in the reachability algorithm depend on this ordering, and therefore the size of the intermediate BDDs. A bad ordering of the partial transition relations can result in intermediate BDDs that are much larger than the final BDD of all reachable configurations. To compute the fixed point using only discrete transitions before applying time transitions is a successful strategy to avoid this problem.

Examples. In the following we report some performance results. All the computation times obtained using our tool are given in seconds of CPU time on a SUN Ultra-Sparc 1 with 200 MHz processor and 64 MB memory. The results of the BDD-based version of Kronos are taken from [BMPY97] and also obtained using a SUN Ultra-Sparc-1.

The BDD-based version of Kronos is able to verify 14 processes of Fischer's mutex protocol. Rabbit needs 13.6 s computation time for the same model. The verification of 32 processes needs 208 s. We are able to verify even the model with 128 of Fischer's processes using 512 MB memory in 9168 s.

Rabbit needs 6 s to compute the whole set of reachable configurations of the AND model with 4 inputs as mentioned in [BMPY97]. Kronos needs 324.7 s for this model. Out tool is able to verify the AND model up to 16 inputs in 1209 s. Another example is the little 'two state' example from [BMPY97], for which they report to handle up to 9 automata. Rabbit computes all rechable configurations of 64 two state automata (having $2.2 \cdot 10^{88}$ configurations) in 94 s. We used on-the-fly analysis for this example, which increases dramatically the performance of models consisting of mostly independent components. For more details about the on-the-fly algorithm see [BN01].

4 Refinement Checking

The intuition behind our refinement concept is an assumption/guarantee principle. We describe it with respect to our formalism: A refinement relation ($\mathcal{P}$ refines $\mathcal{Q}$) for CTA modules $\mathcal{P}$ and $\mathcal{Q}$ has to fulfill the following properties ($\mathcal{M}$ denotes a module, $\mathcal{M}.GI, \mathcal{M}.GO, \mathcal{M}.GMR$ denotes the module's sets of synchronization labels declared as INPUT, OUTPUT, MULTREST, respectively): **(1)** $\mathcal{Q}.GI \subseteq \mathcal{P}.GI$. The occurrence of a synchronization label g as input in a module $\mathcal{M}$ means that $\mathcal{M}$ guarantees that g is not restricted in $\mathcal{M}$. This clearly is a guarantee. Thus each input label of the specification should be an input label of the implementation. **(2)** $\mathcal{P}.GO \subseteq \mathcal{Q}.GO$. The occurrence of a synchronization label g as output in a module $\mathcal{M}$ means that $\mathcal{M}$ assumes that g is not restricted in the environment. The implementation should not make more assumptions than the specification, thus each output label of the implementation should also be an output label in the specification. **(3)** $\mathcal{Q}.G - \mathcal{Q}.GL = \mathcal{P}.G - \mathcal{P}.GL$. The synchronization labels of a module $\mathcal{M}$ can be partitioned into a set of interface labels ($\mathcal{M}.GI \cup \mathcal{M}.GO \cup \mathcal{M}.GMR$), and a set of local labels ($\mathcal{M}.GL$). Interface labels are those via which $\mathcal{M}$ can communicate with the environment. **(4)** $E_{\mathcal{P}} \subseteq E_{\mathcal{Q}}$. The external trace set $E_{\mathcal{P}}$ (defined below) of the labeled transition system generated by $\mathcal{P}$ is a subset of the external trace set $E_{\mathcal{Q}}$ of the labeled transition system generated by $\mathcal{Q}$. The intuition is that the occurrence of a trace t in $E_{\mathcal{Q}}$ means that $E_{\mathcal{Q}}$ *allows* the system behavior t, and the refinement should not allow more behaviors than the specification.

Let $[\![\mathcal{M}]\!]_I = (S, S^0, \mathcal{M}.G \cup \mathbb{N}, \rightarrow_{\mathcal{M}})$ be the labeled transition system generated by module $\mathcal{M}$ (integer semantics). $\mathcal{M}.G$ is the set of synchronization labels of module $\mathcal{M}$, $\mathbb{N}$ is used for the time values and $\mathcal{M}.G \cup \mathbb{N}$ is the set of transition labels of the labeled transition system $[\![\mathcal{M}]\!]_I$. For the exact definition we refer to [BR01]. Roughly spoken, it is the integer semantics of the parallel composition $\mathcal{A}_{\mathcal{M}}$ of all the automata contained by $\mathcal{M}$ regarding various compatibility constraints.

A **trace** of a given labeled transition system $[\![\mathcal{M}]\!]_I$ is an infinite sequence $(a_0, a_1, a_2, \dots)$ of elements of $\mathcal{M}.G \cup \mathbb{N}$. For each element a_k of the trace the following must hold: There exists a run $(s_0, s_1, \dots, s_{k+1}) \in Run([\![\mathcal{M}]\!]_I)$ with $s_i \xrightarrow{a_i}_{\mathcal{M}} s_{i+1}$ for all $i \in \{0, 1, \dots, k\}$.

We use a simulation relation for the algorithmic analysis of refinement within our tool implementation. To define timed simulation we need the notion of external transitions and external traces. After this we can proceed with the algorithm for the simulation check. We use the concept of safety simulation relation as described in [DHWT92].

Let $\tau \in \mathcal{M}.L$ be a local synchronization label. Then a τ-transition $s \xrightarrow{\tau}_{\mathcal{M}} s'$ is called an internal transition. For some configurations s, s' and $a_i \in \mathcal{M}.G \cup \mathbb{N}$ we define $s \xrightarrow{a_1 a_2 \ldots a_n}_{\mathcal{M}} s'$ as: $\exists s''$ with $s \xrightarrow{a_1}_{\mathcal{M}} s''$ and $s'' \xrightarrow{a_2 \ldots a_n}_{\mathcal{M}} s'$. For $a \in (\mathcal{M}.G \setminus \mathcal{M}.L) \cup \mathbb{N}$, $s \xrightarrow{a}_{\mathcal{M}} s'$ is an **external transition** and for $\hat{\tau}, \hat{\tau}' \in (\mathcal{M}.L)^*$, $s \xrightarrow{\hat{\tau} a \hat{\tau}'}_{\mathcal{M}} s'$ is the sequence of an arbitrary number of internal transitions followed by one external transition followed by another arbitrary number of internal transitions. In the sequel we write $s \overset{a}{\rightarrowtail}_{\mathcal{M}} s'$ for $s \xrightarrow{\hat{\tau} a \hat{\tau}'}_{\mathcal{M}} s'$. Now we can define an external trace as follows: The sequence $(a_0, a_1, a_2, \ldots)$ of elements of $(\mathcal{M}.G \setminus \mathcal{M}.L) \cup \mathbb{N}$ is an **external trace**, if for each a_k there exists a sequence $(s_0, s_1, \ldots, s_{k+1}), s_0 \in S^0, s_j \in S, 1 \leq j \leq k+1$ with $s_i \overset{a}{\rightarrowtail}_{\mathcal{M}} s_{i+1}$ for all $i \in \{0, 1, \ldots, k\}$. It hides synchronization labels of internal discrete transitions.

A transition system $\mathcal{Q}$ simulates a transition system $\mathcal{P}$ if $\mathcal{Q}$ can match every step of $\mathcal{P}$ by a step with the same label. We define **timed simulation** for labeled transition systems as follows: The labeled transition system $[\![\mathcal{Q}]\!]_I = (S_{\mathcal{Q}}, S^0_{\mathcal{Q}}, \Sigma_{\mathcal{Q}}, \rightarrow_{\mathcal{Q}})$ simulates the labeled transition system $[\![\mathcal{P}]\!]_I = (S_{\mathcal{P}}, S^0_{\mathcal{P}}, \Sigma_{\mathcal{P}}, \rightarrow_{\mathcal{P}})$, $\Sigma = (\mathcal{P}.G \setminus \mathcal{P}.GL) \cup \mathbb{N} = (\mathcal{Q}.G \setminus \mathcal{Q}.GL) \cup \mathbb{N}, \Sigma_{\mathcal{Q}} = \mathcal{Q}.G \cup \mathbb{N}, \Sigma_{\mathcal{P}} = \mathcal{P}.G \cup \mathbb{N}$, if:

- there exists a simulation relation $\mathcal{R} \subseteq S_{\mathcal{P}} \times S_{\mathcal{Q}}$ which fulfills
  $\forall a \in \Sigma, \forall (p,q) \in \mathcal{R}, \forall p' \in S_{\mathcal{P}}$:
  $\left(p \overset{a}{\rightarrowtail}_{\mathcal{P}} p'\right) \implies \left(\exists q' : q \overset{a}{\rightarrowtail}_{\mathcal{Q}} q' \wedge (p', q') \in \mathcal{R}\right)$, and
- all initial configurations of $\mathcal{P}$ are contained within the simulation relation: $S^0_{\mathcal{P}} \subseteq \{p \in S_{\mathcal{P}} | \exists q : (p,q) \in \mathcal{R}\}$.

The algorithm of the simulation check is shown in Fig. 1. For the composition of $\mathcal{P}$ and $\mathcal{Q}$ we compute the set of reachable configurations. We consider this set of tuples (p, q) as the initial relation for trying to build a simulation relation between $\mathcal{P}$ and $\mathcal{Q}$. Then, in each cycle of a fixed point iteration we assume that it is a simulation relation and we check whether all configurations of the set fulfill the simulation condition from the definition above. Differing from the definition, we use the transition relation of the product $\mathcal{P}||\mathcal{Q}$, which is already computed in our approach. (The computation of the reachable set needs more time than checking the simulation relation if using $\mathcal{P}||\mathcal{Q}$.) If there are 'bad' configurations we have to invalidate our assumption that it is already the simulation relation and we eliminate them from the relation. If we reached the fixed point (i.e. our assumption was true) we got the simulation relation. If the algorithm eliminates some of the initial configurations of $\mathcal{P}$ from the relation there cannot exist a simulation relation and the algorithm aborts. If we reached the fixed point (i.e. our assumption was true) we got the simulation relation.

Note. Modules $\mathcal{P}$ and $\mathcal{Q}$ are not allowed to contain variables within their interfaces (shared variables). The simulation check considers only synchronization labels regarding external traces.

Production Cell Example. To validate the practical relevance of our tool using a complex system, we developed a CTA model of a production cell, which is similar to the Lewerentz/Lindner production cell from FZI. The system consists of 20 machines and belts with 44 sensors and 28 motors. We modeled the system as modular composition of several belts, turntables and machines, including 45 timed automata containing 22

```
Input: labeled transition system $[\![\mathcal{P}]\!]_I = (S_\mathcal{P}, S^0_\mathcal{P}, \Sigma_\mathcal{P}, \rightarrow_\mathcal{P})$
    as integer semantics of module $\mathcal{P}$,
  labeled transition system $[\![\mathcal{P}||\mathcal{Q}]\!]_I = (S_{\mathcal{P}||\mathcal{Q}}, S^0_{\mathcal{P}||\mathcal{Q}}, \Sigma_{\mathcal{P}||\mathcal{Q}}, \rightarrow_{\mathcal{P}||\mathcal{Q}})$
    as integer semantics of the composition $\mathcal{P}||\mathcal{Q}$ with $\Sigma = (\mathcal{P}.G \setminus \mathcal{P}.GL) \cup \mathbb{N}$.
Output: $true$, iff $\mathcal{P}$ simulates $\mathcal{P}||\mathcal{Q}$
$R_{\mathcal{P}||\mathcal{Q}} := Reach([\![\mathcal{P}||\mathcal{Q}]\!]_I)$
do
  $R'_{\mathcal{P}||\mathcal{Q}} := R_{\mathcal{P}||\mathcal{Q}}$
  forall $a \in \Sigma$
    if $S^0_\mathcal{P} \not\subseteq \{p \in S_\mathcal{P} | \exists q : (p,q) \in R_{\mathcal{P}||\mathcal{Q}}\}$ then return false
    $R_{\mathcal{P}||\mathcal{Q}} := R_{\mathcal{P}||\mathcal{Q}} \cap \left\{ (p,q) \in R_{\mathcal{P}||\mathcal{Q}} \;\middle|\; \begin{array}{c} \forall p' : \left(p \overset{a}{\rightarrowtail}_\mathcal{P} p'\right) \Rightarrow \\ \left(\begin{array}{c} \exists q' : (p,q) \overset{a}{\rightarrowtail}_{\mathcal{P}||\mathcal{Q}} (p',q') \\ \wedge \quad (p',q') \in R_{\mathcal{P}||\mathcal{Q}} \end{array}\right) \end{array} \right\}$
while $R_{\mathcal{P}||\mathcal{Q}} \neq R'_{\mathcal{P}||\mathcal{Q}}$
return true
```

Fig. 1. Algorithm for checking a simulation relation

clocks. For the measurement of the throughput, i.e. how long does a piece need to go through the production cycle, we modeled each belt to be able to measure the time of transportation using a clock. For the verification process we can fade out some details of the machines. To verify a safety property, e.g. 'the drilling machine must be off if the transport belt is not off', we verify at first that the timed version of the transport belt implements an untimed version by checking the existence of a simulation relation. Now we can verify the safety property of the model using that smaller untimed version for transport belts. The analysis of the safety property of the system using a timed model for the sensor instances needs 1098 s. Using an untimed version for the transport belts, the same task needs only 556 s. It shows that an abstraction within one small part of the system has a big impact on the computation time. The computation time for the simulation check is 0.5 s because the belt model is a small part of the whole system.

References

[AD94] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[Bey01] Dirk Beyer. Improvements in BDD-based reachability analysis of timed automata. In *Proc. FME 2001*, LNCS 2021, pages 318–343. Springer-Verlag, 2001.

[BMPY97] M. Bozga, O. Maler, A. Pnueli, and S. Yovine. Some progress on the symbolic verification of timed automata. In *Proc. CAV'97*, LNCS 1254, pages 179–190. 1997.

[BN01] Dirk Beyer and Andreas Noack. Efficient verification of timed automata using BDDs. In *Proc. Formal Methods for Industrial Critical Systems*. to appear, 2001.

[BR98] Dirk Beyer and Heinrich Rust. Modeling a production cell as a distributed real-time system with Cottbus Timed Automata. *Proc. FBT'98*, pages 148–159. Shaker, 1998.

[BR01] Dirk Beyer and Heinrich Rust. Cottbus Timed Automata: Formal definition and semantics. In *Proc. FSCBS 2001*, pages 75–87, 2001.

[DHWT92] David L. Dill, Alan J. Hu, and Howard Wong-Toi. Checking for language inclusion using simulation preorders. In *Proc. CAV'91*, LNCS 575, pages 255–265. 1992.