Abstraction from Counters: An Application on Real-Time Systems

G. Logothetis and K. Schneider
Universität Karlsruhe
Institut für Rechnerentwurf und Fehlertoleranz (Prof. Dr.-Ing. D. Schmid)
P.O. Box 6980, 76128 Karlsruhe, Germany
email: {logo,schneide}@informatik.uni-karlsruhe.de
http://goethe.ira.uka.de/hvg

Abstract

We present abstraction techniques for systems containing counters, which allow to significantly reduce their state spaces for their efficient verification. In contrast to previous approaches, our abstraction technique lifts the entire verification problem, i.e., also the specification, to the abstract level.

As an application, we consider the reduction of real-time systems by replacing discrete clocks of timed automata with abstract counters. The presented method allows the reduction of such systems to very small state spaces. As benchmark examples, we consider the generalized railroad crossing and Fischer's mutual exclusion protocol.

1 Introduction

Symbolic model checking [3] has become one of the most successful and reliable methods for formal verification as this verification technique works completely automatic. However, model checking tools suffer from the so-called state explosion problem: This means that the size, i.e. the number of states of the system grows exponentially with the size of the implementation description. There are three typical sources for the state explosion problem:

- The *parallel execution of processes* $P_1, \ldots, P_n$ requires to first determine finite state machines $\mathcal{A}_1, \ldots, \mathcal{A}_n$ for the processes and then to compute the product automaton $\mathcal{A}_\times := \mathcal{A}_1 \times \ldots \times \mathcal{A}_n$. In the worst case, the number of states of $\mathcal{A}_\times$ can become the product of the numbers of states of the automata $\mathcal{A}_i$.
- The *use of real time constraints*: Real-time constraints are often modelled with timed automata. Timed automata extend usual finite state automata by a finite number of clocks. Each time a new state is entered, some of the available clocks can be reset. Transitions leaving a state are moreover guarded by clock constraints that allow or disallow the transition to a successor state.
- Furthermore, systems that have a *non-trivial data flow* suffer from the state explosion due to the representation of the data values by large numbers of boolean variables.

Clearly, there are systems in which the state explosion is caused by more than one factor, like Fischer's mutual exclusion protocol [16] (Section 6), where the state explosion is due to the parallel composition of n processes that additionally have a statement which consumes n time steps. As this is not unusual for real-time systems, the state-explosion problem is even more serious for them: Experiences with automatic verification tools [2, 13, 10] have proven this claim.

To fight the state-explosion problem that stems from complex data values, methods based on different kinds of abstraction are used. These approaches are essentially based on abstract interpretation in the sense of the Cousot's [8]. In abstract interpretation, given an abstraction function and a programming language semantics, we derive an abstract semantics for the programming language, where we replace concrete data types by abstract ones. The basic idea is thereby to apply a predefined abstraction function to obtain an abstract system with a smaller state space. Abstraction of concrete systems to obtain smaller abstract systems is often the only way to make automated finite-state verification procedures applicable to large systems.

Specifications that can be verified in abstract systems have been considered in [7]. Given a system $\mathcal{K}$ over the variables $\mathcal{V}_c$ and an abstraction function $\hbar : \mathcal{V}_c \to \mathcal{V}_a$ that maps concrete variables $\mathcal{V}_c$ to abstract ones $\mathcal{V}_a$, it has been shown how an abstract system $\mathcal{K}_a$ over the concrete variables $\mathcal{V}_c$ can be constructed that preserves any ACTL^* formula over $\mathcal{V}_c$: Whenever it can be proved that $\mathcal{K}_a \models \Phi$ holds, it is guaranteed that also $\mathcal{K}_c \models \Phi$ holds. In case a specification does not hold for the abstract system $\mathcal{K}_a$, nothing can be said about $\mathcal{K}_c \models \Phi$. In many cases, it is however not clear how to construct a suitable abstraction function $\hbar$. Usually it is left up to the verification engineer to find an appropriate abstraction function and to hope that the verification will succeed with the chosen abstraction.

In this paper, we present an abstraction function $\hbar$ that can be applied to a wide range of systems, in particularly to any system that contains counters. Using this abstraction function $\hbar$, our technique constructs from a concrete system $\mathcal{K}_c$ over the concrete variables $\mathcal{V}_c$ an abstract system $\mathcal{K}_\hbar$ over *abstract variables* with less states. Having constructed the abstract system $\mathcal{K}_\hbar$, we furthermore show what kinds of specifications Φ can be lifted to the abstract level, i.e., we construct an *abstract specification* $\Phi_\hbar$. Thus, unlike the approach given in [7], we lift the entire verification problem to the abstract level: Whenever we succeed in verifying that $\mathcal{K}_\hbar \models \Phi_\hbar$ holds, our results imply that also $\mathcal{K} \models \Phi$ holds.

In particular, our method applies to any kind of counters (with increment/decrement inputs) and any kind of ACTL^* specification that compares the counter's value with some fixed constants. The existence of counters in the data flow of a system is a typical reason for state-explosion: It is well-known that counters are complex to handle in symbolic model checking [19, 14, 4]. Counters often arise in many control problems, and in particular the clock constraints in the verification of real-time verification [6] can be modeled by counters. Related work has been done by Macii, Plessier, and Somenzi [19]. They considered a similar approach, which is however limited to special kinds of time-out mechanisms that have only reset and start inputs and do not consider intermediate values of the counter.

The outline of this paper is as follows: In the next Section, we present how we construct abstract systems $\mathcal{K}_\hbar$ and abstract specifications $\Phi_\hbar$ by given abstraction functions $\hbar$. As already noted, our approach allows in contrast to [7] to also lift the specification at the abstract level (cf. definition 3 and theorem 2). In Section 3, we present our abstraction function for counters and refine our criteria for abstract specifications (theorem 3). In Section 4, we present the reduction of timed automata to finite state machines with (abstract) counters so that our abstraction technique can be applied to the verification of real-time systems. In Section 5, we consider as examples the generalized railroad crossing [12], and Fischer's mutual exclusion protocol [16].

2 Construction and Verification of Abstract Systems

We consider systems modeled as Kripke structures over some set of variables V_Σ, i.e., as a tuple $\mathcal{K} = (\mathcal{I}, \mathcal{S}, \mathcal{R}, \mathcal{L})$ where $\mathcal{S}$ is a finite set of states, $\mathcal{I} \subseteq \mathcal{S}$ is the set of initial states, $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{S}$ is the transition relation, and $\mathcal{L}$ is a label function that maps each state $s \in \mathcal{S}$ to a set of variables $\mathcal{L}(s) \subseteq V_\Sigma$. The intension is that the variables $\mathcal{L}(s)$ are those that hold in state s, while the variables in $V_\Sigma \setminus \mathcal{L}(s)$ do not hold in s.

For the specification of a concurrent system, we consider formulas given in the temporal logic ACTL^* [7]. The key idea is that ACTL^* formulas can only express universal quantification over computation paths, i.e., any ACTL^* formula states that some property holds *for all paths of the system.* Abstract systems usually contain more (abstract) paths than the concrete ones, but due to the reduction of the state spaces, it is nevertheless reasonable to reduce systems by means of abstraction.

We write $(\mathcal{K}, s) \models \Phi$ if the ACTL^* formula Φ is satisfied on the state s of the structure $\mathcal{K}$. Moreover, we simply write $\mathcal{K} \models \Phi$ if Φ is satisfied on all initial states of $\mathcal{K}$.

Given a description of a system $\mathcal{P}$ in some programming language, we can derive a corresponding Kripke structure $\mathcal{K}$ for it such that the verification of a temporal logic formula Φ means to check whether $\mathcal{K} \models \Phi$ holds or not. In this paper, we derive directly from $\mathcal{P}$ smaller abstract structures $\mathcal{K}_\hbar$ and abstract specifications $\Phi_\hbar$ such that for any ACTL^* formula Φ, $\mathcal{K}_\hbar \models \Phi_\hbar$ implies $\mathcal{K} \models \Phi$. Given an abstraction function $\hbar$, the corresponding abstract structure $\mathcal{K}_\hbar$ is defined as follows:

Definition 1 (Abstract Structures) *An abstraction function is a function $\hbar : 2^{V_\Sigma} \to 2^{V_\Omega}$ that maps sets of variables of V_Σ to sets of variables of V_Ω. Given a Kripke structure $\mathcal{K} = (\mathcal{I}, \mathcal{S}, \mathcal{R}, \mathcal{L})$ over the variables V_Σ, and an abstraction function $\hbar : 2^{V_\Sigma} \to 2^{V_\Omega}$, we define the abstract Kripke structure $\mathcal{K}_\hbar := (\mathcal{I}_\hbar, \mathcal{S}_\hbar, \mathcal{R}_\hbar, \mathcal{L}_\hbar)$ as follows:*

- $\mathcal{I}_\hbar := \{\hbar(\mathcal{L}(s)) \mid s \in \mathcal{I}\}$
- $\mathcal{S}_\hbar := \{\hbar(\mathcal{L}(s)) \mid s \in \mathcal{S}\}$
- $\mathcal{R}_\hbar := \{(\hbar(\mathcal{L}(s_1)), \hbar(\mathcal{L}(s_2))) \mid (s_1, s_2) \in \mathcal{R}\}$
- $\mathcal{L}_\hbar(\vartheta) := \vartheta$

The relationship between the abstract structure $\mathcal{K}_\hbar$ and the concrete structure $\mathcal{K}$ is as follows: for any path through $\mathcal{K}$ there is a corresponding path through $\mathcal{K}_\hbar$, which means that $\mathcal{K}_\hbar$ can 'simulate' $\mathcal{K}$. This is formally defined as follows:

Definition 2 (Simulation Relations) *Given two structures $\mathcal{K}_1 = (\mathcal{I}_1, \mathcal{S}_1, \mathcal{R}_1, \mathcal{L}_1)$, $\mathcal{K}_2 = (\mathcal{I}_2, \mathcal{S}_2, \mathcal{R}_2, \mathcal{L}_2)$ over variables V_Σ and V_Ω, respectively. A relation σ over $\mathcal{S}_1 \times \mathcal{S}_2$ is called a simulation relation between $\mathcal{K}_1$ and $\mathcal{K}_2$ iff the following holds:*

- **SIM1:** *for states $s_1, s_1' \in \mathcal{S}_1$ and $s_2 \in \mathcal{S}_2$ with $(s_1, s_2) \in \sigma$ and $(s_1, s_1') \in \mathcal{R}_1$, there is a state $s_2' \in \mathcal{S}_2$ such that $(s_2, s_2') \in \mathcal{R}_2$ and $(s_1', s_2') \in \sigma$.*
- **SIM2:** *for any $s_1 \in \mathcal{I}_1$, there is a $s_2 \in \mathcal{I}_2$ with $(s_1, s_2) \in \sigma$.*

$\mathcal{K}_2$ simulates $\mathcal{K}_1$ written as $\mathcal{K}_1 \preceq \mathcal{K}_2$, if there is a simulation relation σ between $\mathcal{K}_1$ and $\mathcal{K}_2$.

It is easy to see that any structure $\mathcal{K}$ is simulated by any abstraction $\mathcal{K}_\hbar$ of it that is constructed as given above. In [18] and [9], it is shown that the above given construction of $\mathcal{K}_\hbar$ is in some sense optimal, i.e. the smallest structure that simulates $\mathcal{K}$. Note that $\mathcal{K}$ does however not necessarily simulate $\mathcal{K}_\hbar$ since the collection of states that have the same labels to a new abstract state can generate new paths that did not appear in the original structure.

Theorem 1 (Simulation of Concrete Structures) *Given a Kripke structure $\mathcal{K} = (\mathcal{I}, \mathcal{S}, \mathcal{R}, \mathcal{L})$ over the variables V_Σ, and an abstraction function $\hbar : 2^{V_\Sigma} \to 2^{V_\Omega}$. Then, the relation $\sigma_\hbar := \{(s, \hbar(\mathcal{L}(s))) \mid s \in \mathcal{S}\}$ is a simulation relation between $\mathcal{K}$ and $\mathcal{K}_\hbar$.*

The proof is straightforward: to prove SIM1, choose $s_2' := \hbar(\mathcal{L}(s_1'))$ and note that $s_2 := \hbar(\mathcal{L}(s_1))$. To prove SIM2, simply use $s_2 := \hbar(\mathcal{L}(s_1))$.

The theorems and definitions we mentioned so far allow us to compute an abstract structure $\mathcal{K}_\hbar$ from a concrete Kripke structure $\mathcal{K}$. It is important to note that by abstract interpretation [8] of the program $\mathcal{P}$ allows us to directly compute the abstract structure $\mathcal{K}_\hbar$ from the program $\mathcal{P}$ without first computing the concrete structure $\mathcal{K}$.

Having computed the abstract structure $\mathcal{K}_\hbar$ from the program, we also need to define an abstraction $\Phi_\hbar$ of the specification Φ such that $\mathcal{K}_\hbar \models \Phi_\hbar$ implies $\mathcal{K} \models \Phi$. This will then enable us to check the simpler problem $\mathcal{K}_\hbar \models \Phi_\hbar$ instead of the more complex problem $\mathcal{K} \models \Phi$.

For this purpose, we have to consider all propositional subformulas of Φ in their disjunctive normal form: let $\mathsf{supp}_{V_\Sigma}(\Phi)$ denote the set of clauses of a propositional formula φ, i.e., $\mathsf{supp}_{V_\Sigma}(\Phi)$ is a set of sets of variables of V_Σ. For example, $\mathsf{supp}_{\{x,y,u\}}((x \wedge \neg y) \vee (\neg x \wedge u)) = \{\{x\}, \{x,u\}, \{y,u\}, \{u\}\}$, i.e. $\mathsf{supp}_{V_\Sigma}(\Phi)$ contains the set of sets of variables that are necessary to make Φ true. Given that $\mathsf{supp}_{V_\Sigma}(\Phi) = \{\vartheta_1, \ldots, \vartheta_n\}$, $\Phi_\hbar$ is obtained as the disjunctive normal form that corresponds with $\{\hbar(\vartheta_1), \ldots, \hbar(\vartheta_n)\}$.

Definition 3 ($\hbar$-Invariant Specifications) *Given a Kripke structure $\mathcal{K} = (\mathcal{I}, \mathcal{S}, \mathcal{R}, \mathcal{L})$ over some set of variables V_Σ, an abstraction function $\hbar : 2^{V_\Sigma} \to 2^{V_\Omega}$, and a ACTL^* formula Φ. Φ is called to be $\hbar$-invariant wrt. $\mathcal{K}$ iff the following holds for all $s \in \mathcal{S}$:*

$$\mathcal{L}(s) \in \mathsf{supp}_{V_\Sigma}(\Phi) \Leftrightarrow \hbar(\mathcal{L}(s)) \in \mathsf{supp}_{V_\Omega}(\mathsf{abs}_\hbar(\Phi))$$

To understand the problem that is solved by $\hbar$-Invariance, consider the following example: Assume, the boolean valued variables $\{x_n, \ldots, x_0\}$ are used to represent some integer number x, and the abstraction function $\hbar$ maps any set ϑ to $\vartheta \setminus \{x_n, \ldots, x_1\}$. Hence, we only consider whether the number x is even or odd. Consider now the specification that amounts to say that for all the time, $x = 10$ must hold. This specification is not invariant under $\hbar$, but the specification that formalizes that x must be an odd or even number is $\hbar$-invariant. For $\hbar$-invariant specifications, we have the following theorem:

Theorem 2 (Abstract Verification) *Given a Kripke structure $\mathcal{K} = (\mathcal{I}, \mathcal{S}, \mathcal{R}, \mathcal{L})$ over some set of variables V_Σ, an abstraction function $\hbar : 2^{V_\Sigma} \to 2^{V_\Omega}$, and an ACTL^* formula Φ that is $\hbar$-invariant wrt. $\mathcal{K}$. Then the following holds:*

$$(\mathcal{K}_\hbar, \hbar(s)) \models \Phi_\hbar \textit{ implies } (\mathcal{K}, s) \models \Phi$$

3 Abstraction of Counters

Choosing appropriate abstraction functions can lead to much smaller abstract structures $\mathcal{K}_\hbar$. In this section, we present our abstraction function $\hbar$ for the abstraction of systems with counters.

Assume $B := \{b_n, \ldots, b_0\} \subseteq V_\Sigma$ is a set of propositional variables that represent an $n+1$-bit unsigned number x with the usual binary representation of natural numbers, i.e. each subset $\vartheta \subseteq B$ represents the natural number $\Theta(\vartheta)$ that is defined as

$$\Theta(\vartheta) := \sum_{i=0}^{n} f_\vartheta(b_i) 2^i \text{ with } f_\vartheta(b_i) = \begin{cases} 1 & : b_i \in \vartheta \\ 0 & : b_i \notin \vartheta \end{cases}$$

Using propositional variables $B := \{b_n, \ldots, b_0\} \subseteq V_\Sigma$, and a finite set of numeric constants[1] $A := \{\alpha_0, \ldots, \alpha_m\} \subseteq \mathbb{N}$, we define the following abstraction function $\hbar_{A,B}$:

Definition 4 (Abstraction Function for Counters) *Given that $B := \{b_n, \ldots, b_0\} \subseteq V_\Sigma$ and some numbers $A := \{\alpha_0, \ldots, \alpha_m\} \subseteq \mathbb{N}$ with $\alpha_i < \alpha_{i+1}$ for $i \in \{0, \ldots, m-1\}$. We define $V_\Omega := (V_\Sigma \setminus B) \cup \{e_0, \ldots, e_m, c_0, \ldots, c_{m+1}\}$ and use then the abstraction function $\hbar_{A,B} : 2^{V_\Sigma} \to 2^{V_\Omega}$ as given in Figure 1.*

$\hbar_{A,B}$ retains all variables of $V_\Sigma \setminus B$, and replaces the variables of $V_\Sigma \cap B$ by singleton sets of the new variables $\{e_0, \ldots, e_m, c_0, \ldots, c_{m+1}\}$, such that e_i means that the current value $\Theta(\vartheta \cap B)$ of x equals to α_i and c_{i+1} means that the current value $\Theta(\vartheta \cap B)$ of x is between α_i and α_{i+1}. In general, whenever the number of representable values, i.e. 2^n is larger than m, the abstract system $\mathcal{K}_{\hbar_{A,B}}$ will be smaller than the original one. Also, note that if n is a generic parameter of the structure $\mathcal{K}$, i.e. if we define for each $n \in \mathbb{N}$ a new structure $\mathcal{K}_n$, then the abstract structure $\mathcal{K}_{\hbar_{A,B}}$ will no longer depend on n. Thus, we can reduce generic and even infinite state spaces to finite ones.

[1]Note that it is not required that each constant $\alpha_i \in A$ satisfies $\alpha_i < 2^{|B|}$.

$$\hbar_{A,B}(\vartheta) := \begin{cases} \{e_i\} \cup (\vartheta \setminus B) & : \Theta(\vartheta \cap B) = \alpha_i \text{ for } i \in \{0,\ldots,m\} \\ \{c_0\} \cup (\vartheta \setminus B) & : \Theta(\vartheta \cap B) < \alpha_0 \\ \{c_{i+1}\} \cup (\vartheta \setminus B) & : \alpha_i < \Theta(\vartheta \cap B) < \alpha_{i+1} \text{ for } i \in \{0,\ldots,m-1\} \\ \{c_{m+1}\} \cup (\vartheta \setminus B) & : \alpha_m < \Theta(\vartheta \cap B) \end{cases}$$

Figure 1. Abstraction function for counters

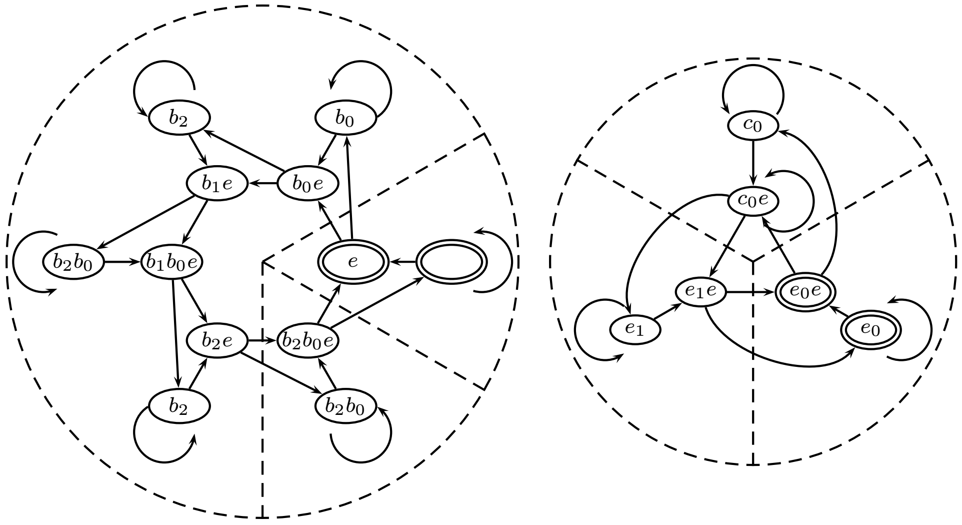

Figure 2. Abstraction of a 3-bit modulo 5 counter

An example is given in Figure 2. The left hand side of Figure 2 gives the Kripke structure of a modulo 5 counter that counts whenever an enable signal e is seen. The counters value is given by the variables $B := \{b_2, b_1, b_0\}$. The right hand side gives the abstract structure $\mathcal{K}_{\hbar_{A,B}}$ that is obtained for the values $A := \{0, 5\}$. In Figure 2, we have given segments of states that correspond to each other, namely the states where the counters value equals to 0 and 5 or is between these numbers. Note that , we can remain infinitely often in the state labeled with c_0e in the abstract structure, but there is no corresponding path in the concrete structure. Hence, the abstract structure is not simulated by the concrete one.

For our abstraction function $\hbar_{A,B}$, this means that we have to assume that the only occurrences of the variables b_i in Φ are comparisons of the 'bitvector $\vec{b} = [b_n, \ldots, b_0]$ with the constants α_i'. To explain this in a bit more detail, assume that for any $\alpha \in \mathbb{N}$, we have propositional formulas $\varphi_{\vec{b}=\alpha}$ and $\psi_{\vec{b}<\alpha}$ over B that evaluate to true under a truth assignment[2] $\vartheta \subseteq B$ iff $\Theta(\vartheta) = \alpha$ and $\Theta(\vartheta) < \alpha$ holds, respectively. In the following, we simply write $\vec{b} = \alpha$ and $\vec{b} < \alpha$ instead of $\varphi_{\vec{b}=\alpha}$ and $\psi_{\vec{b}<\alpha}$, respectively. Under these conditions the following holds:

[2]Evaluating a propositional formula over the variables B with a truth assignment $\vartheta \subseteq B$ means that we consider the variables of ϑ as true and the ones of $B \setminus \vartheta$ as false.

Theorem 3 (Preservation of ACTL* Formulas) *Given a structure $\mathcal{K} = (\mathcal{I}, \mathcal{S}, \mathcal{R}, \mathcal{L})$ over the variables V_Σ, $B := \{b_n, \ldots, b_0\} \subseteq V_\Sigma$, and some numbers $A := \{\alpha_0, \ldots, \alpha_m\} \subseteq \mathbb{N}$ with $\alpha_i < \alpha_{i+1}$ for $i \in \{0, \ldots, m-1\}$. Moreover, let $\Phi \in$ ACTL* be such that any occurrence of $b_i \in B$ in Φ is inside a comparison with one of the constants α_i, i.e. inside a subformula of Φ of one of the following forms: $\vec{b} = \alpha_i$, $\vec{b} < \alpha_i$, or $\alpha_i < \vec{b}$. $\Phi_{\hbar_{A,B}}$ is then obtained by replacing $\vec{b} = \alpha_i$ by e_i, $\vec{b} < \alpha_i$ by $c_i \vee \bigvee_{j=0}^{i-1} c_j \vee e_j$, and $\alpha_i < \vec{b}$ by $c_{m+1} \vee \bigvee_{j=i+1}^{m} c_j \vee e_j$. Then, the following holds:*

$$\mathcal{K}_{\hbar_{A,B}} \models \Phi_{\hbar_{A,B}} \textit{ implies } \mathcal{K} \models \Phi$$

The $\hbar$-invariance is given due to the assumption that the specification does only contain comparisons with the numbers α_i. It is easily seen that under this condition, our specification is indeed $\hbar$-invariant.

4 FSMs with Counters and Timed Automata

Timed automata as given in [1] are frequently used to model real-time systems. Essentially, a timed automaton is a finite state machine that is further equipped with a finite number of clocks. Each clock is a counter that can be reset when a transition is taken. Transitions are enabled iff certain input events are given and additionally a guard, i.e., a condition on the clocks of the system is fulfilled. Guards are equations or inequations where the clock's current values are compared with some real valued constants (the clocks count on a continuous time that is represented by the real numbers). Formally, a timed automaton is defined as follows ($\mathcal{B}(\mathcal{C})$ is the set of boolean combinations of equations and inequations of clocks from $\mathcal{C}$ and real valued constants):

Definition 5 (Timed Automaton) *A timed automaton $\mathcal{A}$ over actions Ack, atomic propositions $\mathcal{P}$, and clocks $\mathcal{C}$ is a tuple $\langle N, l_0, E, V\rangle$. N is a finite set of nodes (control nodes), l_0 is the initial node, $E \subseteq N \times \mathcal{B}(\mathcal{C}) \times Act \times 2^{\mathcal{C}} \times N$ corresponds to the set of transitions, and finally, $V : N \to 2^{\mathcal{P}}$ is a proposition assignment function.*

Configurations of a timed automaton are given with a clock assignment $\xi : \mathcal{C} \to \mathbb{R}$ that maps each clock $\tau \in \mathcal{C}$ to a real number $\xi(\tau)$. A configuration (n, ξ) consists then of such an assignment ξ and a node $n \in N$ for the automaton. If $(n_1, \varphi_{\mathcal{C}}, \alpha, \mathcal{M}_{\mathcal{C}}, n_2) \in E$ holds, and the condition $\varphi_{\mathcal{C}}$ is fulfilled, then a possible successor configuration of (n_1, ξ) is (n_2, ξ'), where ξ' differs from ξ in the values of the clocks $\mathcal{M}_{\mathcal{C}}$ (these are reset to zero).

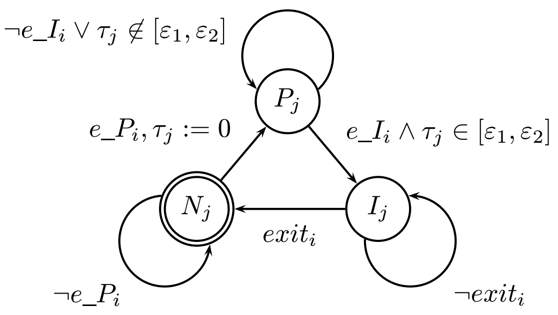
$\neg e\_I_i \vee \tau_j \notin [\varepsilon_1, \varepsilon_2]$
P_j
$e\_P_i, \tau_j := 0$
$e\_I_i \wedge \tau_j \in [\varepsilon_1, \varepsilon_2]$
N_j
I_j
$exit_i$
$\neg e\_P_i$
$\neg exit_i$

Figure 3. The train model

As an example of a timed automaton, consider the diagram given in Figure 3 that models a train in the railway crossing example given in [12]. As given in the next section, a train may be in three regions of a track, namely the region P, the region I, or neither in P nor in I (cf. Figure 4). Figure 3 shows the timed automaton that models a train j. Node N_j means the train is neither in region P nor in region I, while node P_j and I_j mean that the train is in region P and I, respectively. e_P_i and e_I_i are two signals, i.e., propositions of $\mathcal{P}$, emitted whenever a train on track i has entered the region P and I, respectively.

The behavior of the automaton given in Figure 3 is as follows: if the train enters the region P on track i, the signal e_P_i is emitted. Hence, the automaton given in Figure 3 switches to node P_j where a clock τ_j is set to zero. The automaton stays in node P_j until the value of this clock is in the interval $[\varepsilon_1, \varepsilon_2]$. This means that ε_1 and ε_2 are the amounts of time the fastest and slowest train requires to pass region P. We do not count the time the train requires to pass region I, but when this happens, the event $exit_i$ show us this. This will turn the automaton back to node N_j.

It is immediate to see that when we restrict the domain of the clocks of a timed automaton from $\mathbb{R}$ to the natural numbers $\mathbb{N}$, then we can easily model any timed automaton by an ordinary finite state automaton and a finite number of counters. Such a counter can thereby be viewed as a hardware-circuit that increments its value unless it is reset to zero. Additionally, we need comparators for testing the guards that occur in a timed automaton $\mathcal{A}$. The translation of a timed automaton to an ordinary automaton with a couple of counters and comparators is straightforward so that we do not give a formal definition here.

Having constructed a finite-state machine with counters and comparators, we are ready to apply our abstraction function for counters (cf. Figure 1). This reduces the counters together with the comparators to abstract systems with only a few states. Our specification that are given in the temporal logic ACTL* can then easily be checked by any symbolic model checker, i.e., we do not have a need for using real-time verification systems such as UPPAAL [2], KRONOS [10] and VERUS [5].

In the next sections, we list some experimental results we have made with our abstraction technique. We have considered several benchmarks that have often been used by researchers in the domain of real-time verification. In particular, we present our results for the generalized railroad crossing system [12] and Fischer's mutual exclusion protocol [16].

5 Experimental Results

5.1 The Generalized Railroad Crossing

In [12] a benchmark example has been presented for the evaluation of real-time verification tools. This example has been considered in a lot of research papers so far. In this section, we present the application of our abstraction method outlined to this benchmark. The benchmark is a generalized railroad crossing as illustrated in Figure 4.

We first give a short description of the railway crossing. The crossing consists of n tracks (Figure 4 shows only one track) where trains cross a road. Two regions, P and I are distinguished: first a train enters region P which causes the gate $\mathcal{G}$ to start closing. Before the train enters region I, thus leaving region P, the gate must be closed. After the train has left region I, and no other train is neither in region P nor in region I, the gate can be opened again. Note that both

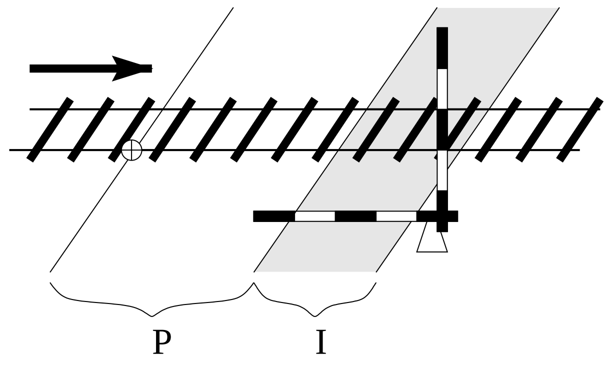

Figure 4. The railroad crossing with one track

the opening and the closing of the gate requires some time and that the system does only work correct when the gate can close faster than the fastest train can pass region P.

This benchmark is a typical example for modeling real-time systems by timed automata [12, 11, 17]. As outlined in the previous section, we can use any model given as timed automaton and transform it first into an ordinary finite-state machine that is additionally equipped with some counters and comparators. Our abstraction then replaces the counters and comparators by abstract ones with only a few states. In the following, we present the abstract model that we obtain by the application of our method to the railway crossing benchmark.

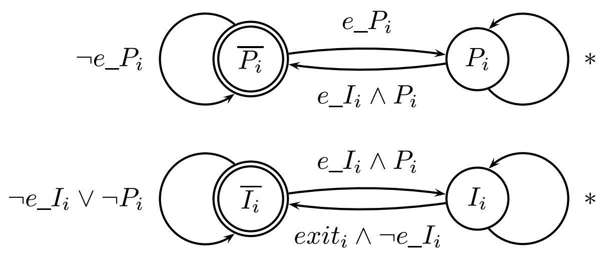

Figure 5. The abstract counters for regions P_i and I_i

Figures 5 and 6 illustrate our approach. Our goal is to model the entire system as one abstract counter, namely the $gate\ \mathcal{G}$ (Figure 6), which uses two additional abstract counters P_i and I_i (Figure 5) for each track i. Counter P_i is incremented (by e_P_i) each time when a train enters region P and decremented ($e\_I_i \wedge \neg e\_P_i$) each time when a train leaves region P to enter region I and no other train is entering region P at the same time. In state $\overline{P_i}$ the number of trains within region P of track i is equal 0 and in state P_i this number is greater than 0.

Counter I_i is incremented (by e_I_i) each time when a train enters region I, thus leaving region P, and is decremented (by $exit \wedge \neg e\_I_i$) each time when a train exits region I and no other train is entering region I at the same time. Note that our method enables the verification of the system for n trains for each gate. Hence, it allows the verification of a *generic number* of trains.

The abstract counters are obtained with our abstraction function for $A := \{0\}$.

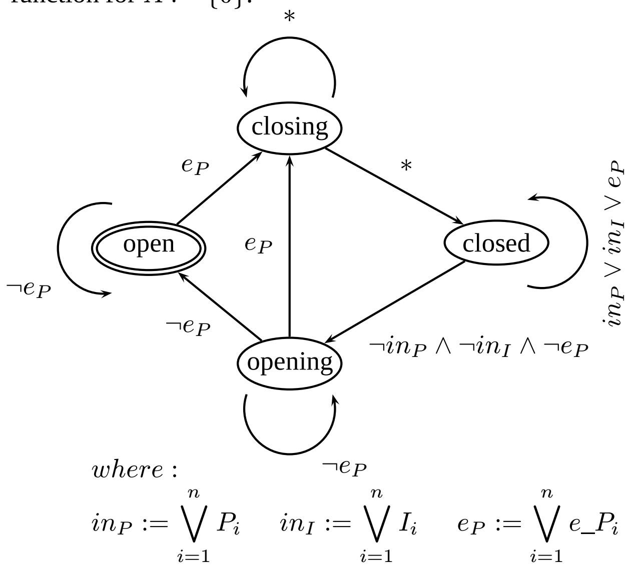
$*$
closing
e_P
$*$
open
e_P
closed
$\neg e_P$
$in_P \vee in_I \vee e_P$
$\neg e_P$
opening
$\neg in_P \wedge \neg in_I \wedge \neg e_P$
$\neg e_P$
$where:$
$in_P := \bigvee_{i=1}^{n} P_i \quad in_I := \bigvee_{i=1}^{n} I_i \quad e_P := \bigvee_{i=1}^{n} e\_P_i$

Figure 6. The gate model as a timed automaton

Initially, the counter (i.e. the gate) $\mathcal{G}$ is in state 'open'. Let γ_{down} and γ_{up} be the durations the gate needs to close and open, respectively. When a train enters region P, the system moves to state 'closing' and starts counting the time. After the time γ_{down} has elapsed, the system moves to state 'closed'. Note that the basic assumption for the correct functioning of the system, is that the gate can close faster than the fastest train can pass the region P. This is given with the following temporal logic formula:

$$\mathsf{G}\left([\mathcal{G}.state = \text{closing}] \rightarrow \neg \bigvee_{i=1}^{n} e\_I_i\right)$$

The counter $\mathcal{G}$ remains in state 'closed' until no trains are in the sections P and I. If this is the case, and additionally no further train is entering at that moment the region P, the system moves to state 'opening' where the gate is opened again, similar to the closing. Note that a further train can enter region P while the system is in 'opening'. If this happens, the opening of the gate has to be interrupted and the controler moves to state ‘closing’ where the gate is about to close again. However, if this does not happen, then finally the gate will be completely opened after time γ_{down} has elapsed. At this point of time, the counter $\mathcal{G}$ moves to state ‘open’.

For the generalized railroad crossing system as described above, the following specifications are to be shown:

$$(1)\quad \mathsf{AG}(in_I \rightarrow \mathcal{G}.state = \text{closed})$$
$$(2)\quad \mathsf{AG}\left(\neg in_P \wedge \neg in_I \rightarrow \mathsf{EF}(\mathcal{G}.state = \text{open})\right)$$

Specification (1) is a safety property that expresses that the gate $\mathcal{G}$ must be closed if a train is in region I. This must hold for all points of time. (2) is a utility property that asserts that the gate can be opened if there is neither a train in region P nor in region I.

We have used the well-known SMV system available from the Carnegie Mellon University to verify both properties for the system with n tracks (note that the number of trains on a track is no longer of interest after our abstraction). The runtime and memory requirements obtained for the properties (1) and (2) on a SUN Ultra 5/10 with 300 MHz and 640 MBytes of main memory that runs under Solaris 7 are given in the tables below.

$\mathsf{AG}(in_I \rightarrow \mathcal{G}.state = \text{closed})$			
Tracks	Run Time [sec]	BDD nodes	reach. States/ poss. States
10	0.13	9624	$2^{50}/2^{53}$
20	0.63	10247	$2^{100}/2^{103}$
30	1.28	11791	$2^{150}/2^{153}$
40	2.19	20716	$2^{200}/2^{203}$
50	3.29	32141	$2^{250}/2^{253}$
60	4.73	46066	$2^{300}/2^{303}$
70	6.37	62491	$2^{350}/2^{353}$
80	8.37	81416	$2^{400}/2^{403}$
90	10.55	102841	$2^{450}/2^{453}$

$\mathsf{AG}\left(\neg in_P \wedge \neg in_I \rightarrow \mathsf{EF}(\mathcal{G}.state = \text{open})\right)$			
Tracks	Run Time [sec]	BDD nodes	reach. States/ poss. States
10	0.14	9609	$2^{51}/2^{52}$
20	0.48	10327	$2^{101}/2^{102}$
30	1.07	11638	$2^{151}/2^{152}$
40	1.75	20513	$2^{201}/2^{202}$
50	2.68	31888	$2^{251}/2^{252}$
60	3.80	45763	$2^{301}/2^{302}$
70	5.15	62138	$2^{351}/2^{352}$
80	6.73	81013	$2^{401}/2^{402}$
90	8.73	102388	$2^{451}/2^{452}$

As a comparison, in [17] the railroad benchmark has been verified as a concrete and as an abstract system as well. The verification of the concrete system was possible for up to 4 tracks and 6 trains per track using the HyTech (runtime 125 s) and the Uppaal (runtime >10 h) tools. The verification of an abstract system was possible for up to 35 tracks using the SMV tool (runtime 3466 s).

As can be seen, our abstraction transforms the benchmark into a quite easily verifiable example for symbolic model checkers which need no longer consider the real-time constraints. However, we have to note that the results given in the above table crucially depends on the variable ordering.

5.2 Fischer’s Mutual Exclusion Protocol

It is well-known that program sections of concurrent processes that modify some shared variables have to be protected to avoid inconsistencies of the data structures. For this reason, several solutions have been suggested. Perhaps the simplest possible algorithm is one suggested by Michael Fischer [16] and is known as *Fischer’s Mutual Exclusion Protocol*. Figure 7 gives some pseudo-code for the protocol: The protocol is used to protect critical sections for δ processes. For this purpose, a global lock variable λ of type $\{0, ..., \delta\}$ is used. The role of λ consists of holding the index of the process that is allowed to enter its critical section. The basic idea of the protocol is roughly as follows (cf. the program code in Figure 7): If $\lambda = 0$ holds, the critical region is currently not owned by a process, so that a process that wants to enter the section can try to obtain access to the region. It therefore will then assign λ its own process id (line s_1). After this, the process will be inactivated for δ units of time so that the other δ processes have the chance to write their process ids to λ. If after that time, λ still contains the process id of the considered process, this process is allowed to enter the critical section and after that, it will release the section by resetting λ to zero.

```
$s_{init}$ :  **repeat**
$s_0$ :           **await** $\lambda = 0$;
$s_1$ :           $\lambda := i$;
$s_2$ :           **sleep** $\delta$;
$s_3$ :       **until** $\lambda = i$;
$s_4$ :       //*critical section*
$s_5$ :       $\lambda := 0$;
```

Figure 7. Fischer’s Mutual Exclusion Protocol

The disadvantage of Fischer’s mutex protocol is that it requires for n processes in each process a delay time δ of order $O(n)$. In the meantime, other solutions have been presented that do not suffer from this disadvantage (cf. [16]).

However, Fischer's mutex protocol is an excellent example to illustrate our method. This example has also been considered by many researchers as a benchmark [15].

We have used McMillan's new implementation of the SMV system to verify Fischer's protocol for n processes without any special composition techniques. The runtime and memory requirements obtained on an Intel Pentium platform with 450 MHz and 512 MByte of main memory that runs under Linux are given in the table below. We verified Fischer's protocol up to 38 processes. To our knowledge, previous results that have been obtained for the automatic verification of the protocol without specialized composition techniques were only able to verify up to 11 processes. Using such techniques, up to 50 processes have been verified in [15].

m	# iter.	BDD nodes (trans.rel.)	BDD nodes (total)	runtime (mutex)
10	7	1997	253850	6.2
20	7	5695	1374366	47.3
30	7	11132	4063029	144.7
38	7	17050	21823704	472.1

The runtimes for the table above have been obtained for an asynchronous version of the protocol, where we assumed that all processes will remain in stage s_2 until all other processes have either moved to s_2 to or have not yet reached s_1. This means that all processes have either written their process id to λ or have not started to do so before any other process will read the value of λ in stage s_3. Moreover, the possible write clashes in stage s_1 have been resolved by nondeterministically choosing one of the values that are concurrently assigned to λ.

References

[1] R. Alur and D. Dill. Automata for modelling real-time systems. *Theoretical Computer Science*, 126(2):183–236, April 1994.

[2] J. Bengtsson, K. Larsen, F. Larsson, P. Pettersson, and W. Yi. UPPAAL in 1995. In *In Proc. of the 2nd Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, number 1055 in Lecture Notes In Computer Science, pages 431–434. Springer-Verlag, March 19956.

[3] J. Burch, E. Clarke, K. McMillan, and D. Dill. Sequential Circuit Verification Using Symbolic Model Checking. In *ACM/IEEE Design Automation Conference (DAC)*, pages 46–51, Los Alamitos, CA, June 1990. ACM/IEEE, IEEE Society Press.

[4] G. Cabodi, P. Camurati, L. Lavagno, and S. Quer. Verification and synthesis of counters based on symbolic techniques. In *European Design and Test Conference (EDTC)*, pages 176–181, Paris, France, March 1997. IEEE Computer Society Press.

[5] S. Campos, E. Clarke, W. Marrero, and M. Minea. Verifying the performance of the PCI local bus using symbolic techniques. Technical Report CMU-CS-96-147, June 1996. ftp://reports.adm.cs.cmu.edu/usr/anon/1996/CMU-CS-96-147.ps.

[6] S. Campos, E. Clarke, W. Marrero, and M. Minea. Verus: A tool for quantitative analysis of finite-state real-time systems. Technical Report CMU-CS-96-159, August 1996.

[7] E. Clarke, O. Grumberg, and D. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and systems*, 16(5):1512–1542, September 1994.

[8] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 238–252. ACM, 1977.

[9] D. Dams, R. Gerth, and O. Grumberg. Abstract interpretation of reactive systems. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1997.

[10] C. Daws, A. Olivero, and S. Yovine. Verifying ET-LOTOS programs with KRONOS. In *In Proc. of 7th International Conference on Formal Description Techniques*, 1994.

[11] 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, TACAS'98*, 1998.

[12] C. Heitmeyer and N. Lynch. The generalize railroad crossing: A case study in formal verification of real-time systems. In *Proc. of the IEEE Real-Time Systems Symposium*, pages 120–131, San Juan, Puerto Rico, December 1994.

[13] P. H. Ho and H. Wong-Toi. Automated analysis of an audio control protocol. In P. Wolper, editor, *Conference on Computer Aided Verification (CAV)*, volume 939 of *Lecture Notes in Computer Science*, pages 381–394, Liege, Belgium, July 1995. Springer Verlag.

[14] S. Krischer. The backward walk approach in FSM verification. In D. Agnew, L. Claesen, and R. Camposano, editors, *IFIP Conference on Computer Hardware Description Languages and their Applications (CHDL)*, pages 143–150, Ottawa, Canada, April 1993. IFIP WG10.2, CHDL'93, IEEE COMPSOC, Elsevier Science Publishers B.V., Amsterdam, Netherland.

[15] K. Kristoffersen, F. Laroussinie, K. Larsen, P. Pettersson, and W. Yi. A compositional proof of a real-time mutual exclusion protocol. In *In Proc. of the 7th International Joint Conference on the Theory and Practice of Software Development*, 1997.

[16] L. Lamport. A fast mutual exclusion algorithm. *ACM Transactions on Computer Systems*, 1987.

[17] A. Loetzbeyer. *Temporale Realzeitverifikation*. PhD thesis, Universitaet Karlsruhe, March 1999.

[18] C. Loiseaux, S. Graf, J. Sifakis, A. Bouajjani, and S. Bensalem. Property preserving abstractions for the verification of concurrent systems. *Formal methods in System Design*, 6:1–35, February 1995.

[19] E. Macii, B. Plessier, and F. Somenzi. Verification of System Containing Counters. In *IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, pages 179–182, Santa Clara, California, November 1992. ACM/IEEE, IEEE Computer Society Press.