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,schneider}@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 \( A_1, \ldots, A_n \) for the processes and then to compute the product automaton \( A_X := A_1 \times \ldots \times A_n \). In the worst case, the number of states of \( A_X \) can become the product of the numbers of states of the automata \( A_i \).

- 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 \( K \) over the variables \( \mathcal{V} \) and an abstraction function \( h : \mathcal{V}_c \to \mathcal{V}_a \) that maps concrete variables \( \mathcal{V}_c \) to abstract ones \( \mathcal{V}_a \), it has been shown how an abstract system \( K_a \) over the concrete variables \( \mathcal{V}_c \) can be constructed that preserves any \( \text{ACTL}^* \) formula over \( \mathcal{V}_c \). Whenever it can be proved that \( K_a \models \Phi \) holds, it is guaranteed that also \( K_c \models \Phi \) holds. In case a specification does not hold for the abstract system \( K_a \), nothing can be
said about $K_c \models \Phi$. In many cases, it is however not clear how to construct a suitable abstraction function $h$. 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 $h$ that can be applied to a wide range of systems, in particular to any system that contains counters. Using this abstraction function $h$, our technique constructs from a concrete system $K_c$ over the concrete variables $V_c$ an abstract system $K_h$ over abstract variables with less states. Having constructed the abstract system $K_h$, we furthermore show what kinds of specifications $\Phi$ can be lifted to the abstract level, i.e., we construct an abstract specification $\Phi_h$. Thus, unlike the approach given in [7], we lift the entire verification problem to the abstract level: Whenever we succeed in verifying that $K_h \models \Phi_h$ holds, our results imply that also $K \models \Phi$ 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 $K_h$ and abstract specifications $\Phi_h$ by given abstraction functions $h$. 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 $V_S$, i.e., as a tuple $K = (I, S, R, L)$ where $S$ is a finite set of states, $I \subseteq S$ is the set of initial states, $R \subseteq S \times S$ is the transition relation, and $L$ is a label function that maps each state $s \in S$ to a set of variables $\mathcal{L}(s) \subseteq V_S$. The intention is that the variables $\mathcal{L}(s)$ are those that hold in state $s$, while the variables in $V_S \setminus \mathcal{L}(s)$ 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 $(K, s) \models \Phi$ if the ACTL* formula $\Phi$ is satisfied on the state $s$ of the structure $K$. Moreover, we simply write $K \models \Phi$ if $\Phi$ is satisfied on all initial states of $K$.

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

**Definition 1 (Abstract Structures)** An abstraction function is a function $h : 2^{V_S} \rightarrow 2^{V_h}$ that maps sets of variables of $V_S$ to sets of variables of $V_h$. Given a Kripke structure $K = (I, S, R, L)$ over the variables $V_S$, and an abstraction function $h : 2^{V_S} \rightarrow 2^{V_h}$, we define the abstract Kripke structure $K_h := (h(I), h(S), h(R), h(L))$ as follows:

- $h(I)$ is the transition relation, and $h(L)$ is a label function that maps each state $s \in h(S)$ to a set of variables $h(\mathcal{L}(s)) \subseteq h(V_S)$. The intention is that the variables $h(\mathcal{L}(s))$ are those that hold in state $s$, while the variables in $h(V_S) \setminus h(\mathcal{L}(s))$ 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 $(K, s) \models \Phi$ if the ACTL* formula $\Phi$ is satisfied on the state $s$ of the structure $K$. Moreover, we simply write $K \models \Phi$ if $\Phi$ is satisfied on all initial states of $K$.

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

**Definition 2 (Simulation Relations)** Given two structures $K_1 = (I_1, S_1, R_1, L_1)$, $K_2 = (I_2, S_2, R_2, L_2)$ over variables $V_{S\Xi}$ and $V_{\Omega}$, respectively. A relation $\sigma$ over $S_1 \times S_2$ is called a simulation relation between $K_1$ and $K_2$ iff the following holds:

- **SIM1**: for states $s_1, s'_1 \in S_1$ and $s_2 \in S_2$ with $(s_1, s_2) \in \sigma$ and $s'_1 \in R_1$, there is a state $s'_2 \in S_2$ such that $(s_2, s'_2) \in R_2$ and $(s'_1, s'_2) \in \sigma$.

- **SIM2**: for any $s_1 \in I_1$, there is a $s_2 \in I_2$ with $(s_1, s_2) \in \sigma$.

$K_2$ simulates $K_1$ written as $K_1 \preceq K_2$, if there is a simulation relation $\sigma$ between $K_1$ and $K_2$. 
It is easy to see that any structure \( \mathcal{K} \) is simulated by any abstraction \( \mathcal{K}_h \) of it that is constructed as given above. In [18] and [9], it is shown that the above given construction of \( \mathcal{K}_h \) is in some sense optimal, i.e., the smallest structure that simulates \( \mathcal{K} \). Note that \( \mathcal{K} \) does however not necessarily simulate \( \mathcal{K}_h \) 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 \( V_\Sigma \), and an abstraction function \( h : 2^{V_\Sigma} \rightarrow 2^{V_0} \). Then, the relation \( \sigma_h := \{(s, h(L(s))) | s \in \mathcal{S}\} \) is a simulation relation between \( \mathcal{K} \) and \( \mathcal{K}_h \).

The proof is straightforward: to prove SIM1, choose \( s'_2 := h(L(s_1)) \) and note that \( s_2 := h(L(s_1)) \). To prove SIM2, simply use \( s_2 := h(L(s_1)) \).

The theorems and definitions we mentioned so far allow us to compute an abstract structure \( \mathcal{K}_h \) 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 \( \mathcal{K}_h \) from the program \( \mathcal{P} \) without first computing the concrete structure \( \mathcal{K} \).

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

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

**Definition 3 (h-Invariant Specifications)** Given a Kripke structure \( \mathcal{K} = (\mathcal{I}, \mathcal{S}, \mathcal{R}, \mathcal{L}) \) over some set of variables \( V_\Sigma \), an abstraction function \( h : 2^{V_\Sigma} \rightarrow 2^{V_0} \), and an ACTL* formula \( \Phi \). \( \Phi \) is called to be h-invariant wrt. \( \mathcal{K} \) iff the following holds for all \( s \in \mathcal{S} \):

\[
\mathcal{L}(s) \in \text{supp}_{V_\Sigma}(\Phi) \iff h(L(s)) \in \text{supp}_{V_0}(\text{abs}_h(\Phi))
\]

To understand the problem that is solved by \( h \)-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 \( h \) maps any set \( \vartheta \) 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 \( h \), but the specification that formalizes that \( x \) must be an odd or even number is \( h \)-invariant. For \( h \)-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 \( V_\Sigma \), an abstraction function \( h : 2^{V_\Sigma} \rightarrow 2^{V_0} \), and an ACTL* formula \( \Phi \) that is \( h \)-invariant wrt. \( \mathcal{K} \). Then the following holds:

\[
(\mathcal{K}_h, h(s)) \models \Phi_h \text{ implies } (\mathcal{K}, s) \models \Phi
\]

### 3 Abstraction of Counters

Choosing appropriate abstraction functions can lead to much smaller abstract structures \( \mathcal{K}_h \). In this section, we present our abstraction function \( h \) 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

\[
\Theta(\vartheta) := \sum_{i=0}^{n} f_\alpha(b_i)2^i \text{ with } f_\alpha(b_i) = \left\{ \begin{array}{ll} 1: & b_i \in \vartheta \\ 0: & b_i \notin \vartheta \end{array} \right.
\]

Using propositional variables \( B := \{b_n, \ldots, b_0\} \subseteq V_\Sigma \), and a finite set of numeric constants\(^1\) \( A := \{a_0, \ldots, a_m\} \subseteq \mathbb{N} \), we define the following abstraction function \( h_{A,B} \):

**Definition 4 (Abstraction Function for Counters)** Given that \( B := \{b_n, \ldots, b_0\} \subseteq V_\Sigma \) and some numbers \( A := \{a_0, \ldots, a_m\} \subseteq \mathbb{N} \) with \( a_i < a_{i+1} \) for \( i \in \{0, \ldots, m-1\} \),

We define \( V_\Omega := \langle V_\Sigma \setminus B \rangle \cup \{c_0, \ldots, c_m, c_0, \ldots, c_{m+1}\} \) and use then the abstraction function \( h_{A,B} : 2^{V_\Sigma} \rightarrow 2^{V_0} \) as given in Figure 1.

\( h_{A,B} \) retains all variables of \( V_\Sigma \setminus B \), and replaces the variables of \( V_\Sigma \cap B \) by singleton sets of the new variables \( \{c_0, \ldots, c_m, c_0, \ldots, c_{m+1}\} \), such that \( c_i \) means that the current value \( \Theta(\vartheta \cap B) \) of \( x \) equals to \( a_i \) and \( c_{i+1} \) means that the current value \( \Theta(\vartheta \cap B) \) of \( x \) is between \( a_i \) and \( a_{i+1} \). In general, whenever the number of representable values, i.e. \( 2^n \) is larger than \( m \), the abstract system \( \mathcal{K}_{h_{A,B}} \) 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 \( \mathcal{K}_{h_{A,B}} \) 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 \( a_i \in A \) satisfies \( a_i < 2^{10} \).
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 $K_{h_1} \ (A, B)$ 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.

Figure 1. Abstraction function for counters

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

For our abstraction function $h_{A,B}$, this means that we have to assume that the only occurrences of the variables $b_i$ in $\Phi$ are comparisons of the “bitvector $\vec{b} = [b_{n_1}, \ldots, b_0]$ with the constants $\alpha_i$”. 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 $\vartheta \subseteq B$ 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:

**Theorem 3 (Preservation of ACTL* Formulas)** Given a structure $K = (I, S, R, L)$ over the variables $V_S$, $B := \{b_{n_1}, \ldots, b_0\} \subseteq V_S$, 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 $\Phi \in \text{ACTL}^*$ be such that any occurrence of $b_i \in B$ in $\Phi$ is inside a comparison with one of the constants $\alpha_i$, i.e., inside a subformula of $\Phi$ of one of the following forms: $\vec{b} = \alpha_i$, $\vec{b} < \alpha_i$, or $\vec{b} \leq \alpha_i$. $\Phi_{h_{A,B}}$ is then obtained by replacing $\vec{b} = \alpha_i$ by $e_i$, $\vec{b} < \alpha_i$ by $e_i \lor \bigvee_{j=0}^{i-1} c_j \lor e_j$, and $\vec{b} \leq \alpha_i$ by $c_{i+1} \lor \bigvee_{j=i+1}^{m} c_j \lor e_j$. Then, the following holds:

$$K_{h_{A,B}} \models \Phi_{h_{A,B}} \text{ implies } K \models \Phi$$

The $h$-invariance is given due to the assumption that the specification does only contain comparisons with the numbers $\alpha_i$. It is easily seen that under this condition, our specification is indeed $h$-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 $\langle B(C), \tau \rangle$ is the set of boolean combinations of equations and inequations of clocks from $C$ and real valued constants):

**Definition 5 (Timed Automaton)** A timed automaton $A$ over actions $\text{Act}$, atomic propositions $P$, and clocks $C$ is a tuple $\langle N, I_0, E, \forall \rangle$. $N$ is a finite set of nodes (control nodes), $I_0$ is the initial node, $E \subseteq N \times B(C) \times \text{Act} \times 2^P$, and $\forall$ is a proposition assignment function.

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

$$\neg e_{-P_i} \lor \tau_j \notin [\varepsilon_1, \varepsilon_2]$$

$$e_{-P_i}, \tau_j := 0$$

$$e_{-I_i} \land \tau_j \in [\varepsilon_1, \varepsilon_2]$$

![Figure 3. The train model](image)

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 $P$, 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 $\tau_j$ 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 $\varepsilon_1$ and $\varepsilon_2$ 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 $\text{exit}_i$ 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 $A$. 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 $\text{ACTL}^*$ 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
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.

Figures 5 and 6 illustrate our approach. Our goal is to model the entire system as one abstract counter, namely the gate \( G \) (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 (by \( e_{I_i \land P_i} \)) 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 \( T_i \) 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 \( exit_i \land \neg e_{-I_i} \)) 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 \( A := \{0\} \).

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

\[
G \left( [G.\text{state} = \text{closing}] \rightarrow \neg \bigvee_{i=1}^n e_{-I_i} \right)
\]

The counter \( G \) 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
controller moves to state ‘closing’ where the gate is about
to close again. However, if this does not happen, then fi-
nally the gate will be completely opened after time $\gamma_{down}$
has elapsed. At this point of time, the counter $G$ moves to
state ‘open’.

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

\begin{align*}
(1) & \quad AG(in_f \rightarrow G.state = \text{closed}) \\
(2) & \quad AG (\neg in_P \land \neg in_f \rightarrow EF(G.state = \text{open}))
\end{align*}

Specification (1) is a safety property that expresses that the
gate $G$ 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 prop-
erties for the system with $n$ tracks (note that the number of
trains on a track is no longer of interest after our abstrac-
tion). 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 So-
laris 7 are given in the tables below.

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|}
\hline
Tracks & Run Time & BDD nodes & reach. States/ poss. States \\
\hline
10 & 0.13 & 9624 & $2^{30}/2^{23}$ \\
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 & $2^{450}/2^{453}$ \\
\hline
\end{tabular}
\caption{AG(in_f \rightarrow G.state = \text{closed})}
\end{table}

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|}
\hline
Tracks & Run Time & BDD nodes & reach. States/ poss. States \\
\hline
10 & 0.14 & 9609 & $2^{31}/2^{22}$ \\
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 & $2^{451}/2^{452}$ \\
\hline
\end{tabular}
\caption{AG (\neg in_P \land \neg in_f \rightarrow EF(G.state = \text{open}))}
\end{table}

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 bench-
mark 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.

\section{5.2 Fischer’s Mutual Exclusion Protocol}

It is well-known that program sections of concurrent pro-
cesses that modify some shared variables have to be pro-
tected 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 \textit{Fischer’s Mutual Exclusion
Protocol}. Figure 7 gives some pseudo-code for the proto-
col: The protocol is used to protect critical sections for $\delta$
processes. For this purpose, a global lock variable $\lambda$ of type
$\{0, \ldots, \delta\}$ is used. The role of $\lambda$ 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 re-
gion 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 $\lambda$ its own process
id (line $s_1$). After this, the process will be inactivated for $\delta$
units of time so that the other $\delta$ processes have the chance
to write their process ids to $\lambda$. If after that time, $\lambda$ still con-
tains 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 $\lambda$ to zero.

\begin{verbatim}
s_init : repeat
s_0 : await $\lambda = 0$
\end{verbatim}

\begin{verbatim}
s_1 : $\lambda := i$
\end{verbatim}

\begin{verbatim}
s_2 : sleep $\delta$
\end{verbatim}

\begin{verbatim}
s_3 : until $\lambda = i$
\end{verbatim}

\begin{verbatim}
s_4 :  //critical section
\end{verbatim}

\begin{verbatim}
s_5 : $\lambda := 0$
\end{verbatim}

\begin{figure}[h]
\centering
\includegraphics{fig7.png}
\caption{Fischer’s Mutual Exclusion Protocol}
\end{figure}

The disadvantage of Fischer’s mutex protocol is that it
requires for $n$ processes in each process a delay time $\delta$ of
order $O(n)$. In the meantime, other solutions have been pre-
sented 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].

<table>
<thead>
<tr>
<th>$m$</th>
<th># iter.</th>
<th>BDD nodes (trans.rel.)</th>
<th>BDD nodes (total)</th>
<th>runtime (mutex)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>7</td>
<td>1997</td>
<td>253850</td>
<td>6.2</td>
</tr>
<tr>
<td>20</td>
<td>7</td>
<td>5695</td>
<td>1374366</td>
<td>47.3</td>
</tr>
<tr>
<td>30</td>
<td>7</td>
<td>11132</td>
<td>4063029</td>
<td>144.7</td>
</tr>
<tr>
<td>38</td>
<td>7</td>
<td>17050</td>
<td>21823704</td>
<td>472.1</td>
</tr>
</tbody>
</table>

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 $s_2$ or have not yet reached $s_1$. This means that all processes have either written their process id to $\lambda$ or have not started to do so before any other process will read the value of $\lambda$ 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 $\lambda$.

References


