Abstraction of Systems with Counters for Symbolic Model Checking*

Klaus Schneider and George Logothetis

University of Karlsruhe, Department of Computer Science, Institute for Computer Design and Fault Tolerance (Prof. Dr.-Ing. D. Schmid), P.O. Box 6980, 76128 Karlsruhe, Germany, email: {Klaus.Schneider,George.Logothetis}@informatik.uni-karlsruhe.de, http://goethe.ira.uka.de/

Abstract Model checking of temporal logics has become a standard technique for the verification of finite state reactive systems. However, these procedures suffer from the so-called state explosion problem which limits their practical use. Therefore, appropriate abstractions have to be applied to reduce the state space if these tools are to be applied to real-world problems. In particular, counters are hard to verify with model checking procedures. Hence, we present in this paper a special abstraction technique for counters that leads to very small, and in particular finite, state spaces. The method even allows in many cases to verify generic systems without interactive theorem proving, i.e. without induction. As counters are often used for the implementation of control systems, the method presented here is of essential importance for the verification of these systems.

1 Introduction

Modern reactive systems are becoming very complicated, so that the application of methods to ensure their correctness is mandatory. Formal verification techniques like model checking and theorem proving are the most reliable methods to guarantee the correctness of the designs. However, each of these approaches to the verification problem has its own advantages and disadvantages: while theorem proving can handle generic systems with abstract data types, it can be applied to large systems as e.g. microprocessors. On the other hand, theorem proving is done – to a large extent – in an interactive manner which needs a lot of experience to perform the verification task quickly.

Model checking techniques, on the other hand, work completely automatic, but 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. In particular, systems that have a non-trivial data flow suffer from the state explosion, while usually control flow dominated systems do only consist of a moderate number of states. This is why it is sometimes stated that model checking lends itself only for the verification of control flow properties.

In particular, counters in the data flow are hard to verify by symbolic model checking [1,2,3], but do often arise in many control problems, and in particular in the verification of real-time verification [4]. To fight the state-explosion problem that arises

* This work has been financed by the DFG priority program ‘Design and Design Methodology of Embedded Systems’.
from complex data paths, enhanced model checking procedures have been developed
that abstract from the data types that occur in the system. These approaches are essen-
tially based on abstract interpretation [5]. In abstract interpretation, given an abstraction
function and a programming language semantics, we derive an abstract semantics for
the programming language, where we replace the concrete data types by abstract ones.

The basic idea is thereby is to apply a predefined abstraction function to obtain an
abstract system with a smaller state space. When abstraction is applied to data types,
it is even possible to reduce infinite state spaces to finite ones. This makes finite state
approaches as model checking applicable to the verification of such systems. Also,
some systems are generic, e.g. in terms of the width of registers or other parameters.
Abstractions can sometimes be applied such that the specific value of the parameters is
no longer of interest, so that a verification of the abstract system proves the correctness
of all instances of the parameters without using induction or other means of theorem
proving methods.

Specifications that can be verified in the abstract system have been considered in [6].
There, a special temporal logic $\text{ACTL}^*$ has been defined such that whenever a $\text{ACTL}^*$
specification can be proved for the abstract system, we know that it does also hold for
the concrete system. In case a specification does not hold for the abstract system, we
can however say nothing about the concrete system.

In many cases, it is however not clear how the abstraction should be made. 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. For some examples,
it is however almost obvious what kind of abstraction should be applied.

As such an example, we present in this paper an abstraction function that can be
applied to any system that contains counters. The technique dramatically reduces the
number of states such that model checking can often be effectively used for its verifi-
cation. Our method applies to any kind of counters (with increment/decrement inputs)
and any kind of LTL specification that compares the counter’s value only with some
fixed constants.

Somenzi et al. considered in [1] a similar approach. His method is however limited
to a special kind of timeout counter that has only a reset and a start input. Moreover, [1]
does not consider intermediate values of the counter.

The outline of this paper is as follows: in the next section, we present the basics of
the abstraction for the verification of reactive systems. Moreover, we define a general
abstraction function for counters. Section 3 illustrates the method in detail by means
of a case study and in section 4, we give an overview on future work to extend the
application to other systems.

2 The Abstraction Function for Counters

We consider systems given as Kripke structures over some set of variables $V_\Sigma$, i.e. as
a tuple $\mathcal{K} = (I, S, R, L)$ where $S$ is a 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 $L(s) \subseteq V_\Sigma$. The intension is that the variables $L(s)$ are
those that hold in state $s$, while the variables in $V_\Sigma \setminus L(s)$ do not hold in $s$. We use the
temporal logic LTL as formalism for our specifications.
Definition 1 (Linear Temporal Logic). Given a set of variables \( \Sigma \), the set of \( \text{LTL}_{\Sigma} \) formulas is the smallest set that satisfies the following rules:

- \( x \in \Sigma \) implies \( x \in \text{LTL}_{\Sigma} \)
- \( \varphi, \psi \in \text{LTL}_{\Sigma} \) implies \( \neg \varphi, \varphi \land \psi, \varphi \lor \psi \in \text{LTL}_{\Sigma} \)
- \( \varphi, \psi \in \text{LTL}_{\Sigma} \) implies \( \exists \varphi, \forall \varphi, [\varphi \lor \psi] \in \text{LTL}_{\Sigma} \)

The semantics of \( \text{LTL}_{\Sigma} \) formulas is defined wrt. a Kripke structure over the same set of variables \( \Sigma \) (see e.g. [7]). We write \( \mathcal{K}, \pi \models \Phi \) if the \( \text{LTL}_{\Sigma} \) formula \( \Phi \) is satisfied on the path \( \pi \) of the structure \( \mathcal{K} \) and \( \mathcal{K} \models \Phi \) if \( \Phi \) is satisfied on all paths of the structure \( \mathcal{K} \) that start in an initial state.

Given a circuit, we can derive a corresponding Kripke structure \( \mathcal{K} \) for it such that the verification of a \( \text{LTL}_{\Sigma} \) formula \( \Phi \) means to check whether \( \mathcal{K} \models \Phi \) holds or not. In this paper, we aim however at computing smaller structures \( \mathcal{K}_h \) and abstract specifications \( \Phi_h \) such that for any \( \text{LTL}_{\Sigma} \) formula \( \Phi \), \( \mathcal{K}_h \models \Phi_h \) implies \( \mathcal{K} \models \Phi \). The definition of such an abstract structure \( \mathcal{K}_h \) is as follows[1].

Definition 2 (Abstract Structures). An abstraction function is a function \( h : 2^\Sigma \to 2^\Omega \) that maps sets of variables of \( \Sigma \) to sets of variables of \( \Omega \). Given a Kripke structure \( \mathcal{K} = (I, S, R, \mathcal{L}) \) over the variables \( \Sigma \), and an abstraction function \( h : 2^\Sigma \to 2^\Omega \), we define the abstract Kripke structure \( \mathcal{K}_h := (I_h, S_h, R_h, \mathcal{L}_h) \) where \( I_h := \{ h(\mathcal{L}(s)) \mid s \in I \} \), \( S_h := \{ h(\mathcal{L}(s)) \mid s \in S \} \), \( R_h := \{ (h(\mathcal{L}(s_1)), h(\mathcal{L}(s_2))) \mid (s_1, s_2) \in R \} \), and \( \mathcal{L}_h(\varnothing) := \varnothing \).

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

Definition 3 (Simulation Relations). Given two structures \( \mathcal{K}_1 = (I_1, S_1, R_1, \mathcal{L}_1) \), \( \mathcal{K}_2 = (I_2, S_2, R_2, \mathcal{L}_2) \) over variables \( \Sigma \) and \( \Omega \), respectively. A relation \( \sigma \) over \( S_1 \times S_2 \) is called a simulation relation between \( \mathcal{K}_1 \) and \( \mathcal{K}_2 \) iff the following holds:

- **SIM1**: for states \( s_1, s'_1 \in S_1 \) and \( s_2, s'_2 \in S_2 \) with \( (s_1, s_2) \in \sigma \) and \( (s_1, s'_1) \in R_1 \), 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 \( (s_1, s_2) \in \sigma \).

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

It is easy to see that any structure is simulated by any abstraction of it. The converse does however not necessarily hold 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 4 (Simulation of Concrete Structures). Given a Kripke structure \( \mathcal{K} = (I, S, R, \mathcal{L}) \) over the variables \( \Sigma \), and an abstraction function \( h : 2^\Sigma \to 2^\Omega \). Then, the relation \( \sigma_h := \{ (s, h(\mathcal{L}(s))) \mid s \in S \} \) is a simulation relation between \( \mathcal{K} \) and \( \mathcal{K}_h \).

---

1 The abstract structure is obtained as a quotient with the equivalence relation \( s_1 \sim s_2 :\iff h(\mathcal{L}(s_1)) = h(\mathcal{L}(s_2)) \). States of \( \mathcal{K}_h \) are simple the equivalence classes.
The proof is straightforward: to prove SIM1, choose \( s'_2 := h(\mathcal{L}(s'_1)) \) and note that \( s_2 := h(\mathcal{L}(s_1)) \). To prove SIM2, simply use \( s_2 := h(\mathcal{L}(s_1)) \).

Choosing appropriate abstraction functions can lead to much smaller abstract structures \( \mathcal{K}_h \). For example, assume \( B := \{b_n, \ldots, b_0\} \subseteq V_\Sigma \) is a set of variables that represent an \( n + 1 \)-bit unsigned number 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 a finite set of constants \( A \), we define the following abstraction function \( h_{A,B} \):

**Definition 5 (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 the following abstraction function \( h_{A,B} : 2^{V_\Sigma} \to 2^{V_\Omega} \):

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

\( h_{A,B} \) retains all variables of \( V_\Sigma \setminus B \), and replaces the variables of \( V_\Sigma \cap B \) by singleton sets of the new variables \( \{e_0, \ldots, e_m, c_0, \ldots, c_{m+1}\} \), such that \( e_i \) means that \( \Theta(\vartheta \cap B) \) equals to \( \alpha_i \) and \( c_{i+1} \) means that \( \Theta(\vartheta \cap B) \) is between \( \alpha_i \) and \( \alpha_{i+1} \). To see that the abstraction leads to smaller state spaces, assume that \( B = V_\Sigma \). Note that the abstract structure will then have at most \( 2m + 3 \) different states, while the given one may have \( 2^n \) different labels on its states. In general, whenever the number of representable numbers, i.e. \( 2^n \) is larger than \( m \), the abstract system \( \mathcal{K}_{h_{A,B}} \) will be significantly 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.

An example is given in figure [1]. The left hand side of figure [1] 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 \( \mathcal{K}_{h_{A,B}} \) that is obtained for the values \( A := \{0, 5\} \). In figure [1] 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 \( c_0 e \) 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.

The theorems and definitions we mentioned so far allow us to compute an abstract structure from a concrete Kripke structure. It is important to note that abstract interpretation [3] of the hardware description allows us to directly compute the abstract structure from the hardware description without first computing the concrete structure.

\[2\] Note that it is not required that each constant \( \alpha_i \in A \) satisfies \( \alpha_i < 2^{|B|} \).
Having computed the abstract structure $\mathcal{K}_h$ from the circuits description, 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 $\mathcal{K} \models \Phi$.

The problem is that our specification $\Phi$ must be consistent with our abstraction function $h$. 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 $b = [b_n, \ldots, b_0]$ 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 $\varphi_{b=\alpha}$ and $\psi_{b<\alpha}$ over $B$ that evaluate to true under a truth assignment $\theta \subseteq B$ iff $\Theta(\theta) = \alpha$ and $\Theta(\theta) < \alpha$ holds, respectively. In the following, we simply write $b = \alpha$ and $b < \alpha$ instead of $\varphi_{b=\alpha}$ and $\psi_{b<\alpha}$, respectively. Under these conditions the following holds:

**Theorem 6 (Preservation of LTL Formulas).** Given a structure $\mathcal{K} = (I, S, R, \mathcal{L})$ over the variables $V_\Sigma$, $B := \{b_n, \ldots, b_0\} \subseteq V_\Sigma$, 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 $\Phi \in \text{LTL}_V$ 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: $b = \alpha_i$, $b < \alpha_i$, or $\alpha_i < b$.

$\Phi_{h_{A,B}}$ is then obtained by replacing $b = \alpha_i$ by $e_i$, $b < \alpha_i$ by $c_i \lor \lor_{j=0}^{i-1} c_j \lor e_j$, and $\alpha_i < b$ by $c_{m+1} \lor \lor_{j=i+1}^{m} c_j \lor e_j$. Then, the following holds:

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

In fact, the above theorem can be extended to ACTL$^*$ specifications, but we are only interested in the logic LTL. Note that the other direction does not hold: we can not conclude from $\mathcal{K}_{h_{A,B}} \not\models \Phi_{h_{A,B}}$ that $\mathcal{K} \not\models \Phi$ holds.

---

3 Evaluating a propositional formula over the variables $B$ with a truth assignment $\theta \subseteq B$ means that we consider the variables of $\theta$ as true and the ones of $B \setminus \theta$ as false.
3 A Case Study: The Island Traffic Control Problem

In this section, we apply the method presented in the previous section to a case study to illustrate its practicability. The example considered in this section has already been considered under several aspects [8,9].

3.1 The Implementation

The basic task of the system is to control the traffic to and from an island that is connected with the mainland with a tunnel (see figure 2). The problem is that the street in the tunnel is too narrow that cars can pass it in both directions at the same time. Hence, the control of the tunnel must either grant the cars to travel from the mainland to the island or vice versa, but never both at the same time. Additionally, there is the restriction that the number of cars that are allowed to be on the island is limited by a maximal number $\hat{n}$.

The controller has the task to control the traffic lights at the each end of this tunnel such that both traffic lights should never have a green light at the same time. Moreover, the system must be always alive, i.e. if a car wants to enter the tunnel, it may do so after a finite amount of time. Furthermore, the limitation of the number of cars on the island is respected: at no point of time there should be more than $\hat{n}$ cars on the island. To guarantee the liveness, we must of course assume that not all cars that are on the island will remain there forever.

There are four sensors for detecting the presence of vehicles: one at the tunnel entrance on the island side $il_{\text{enter}}$, one at the tunnel exit on the island side $il_{\text{exit}}$, one at the tunnel entrance on the mainland side $ml_{\text{enter}}$, and one at the tunnel exit on the mainland side $ml_{\text{exit}}$ (see figure 2).

An implementation of the tunnel controller was given in [8,9] and involves three processes implemented by three subcontrollers and two counters for counting the number of cars presently inside the tunnel ($TC$) and the number of cars presently on the island ($IC$). As subcontrollers there are two side controllers and a tunnel access controller. The latter is an arbiter which grants the control of the traffic, i.e. the control over the tunnel, either the mainland or the island controller. If the control of the traffic should be transferred from the mainland to the island controller or vice versa, then we first need a phase where both the mainland and the island controller are disabled. In this phase, both traffic lights have a red light as long as cars are inside the tunnel. If the last car leaves the tunnel, then the control of the tunnel can pass to either the mainland or the island controller. Hence, we see that here it is important to check whether $TC = 0$ holds or not.

The mainland controller is only given the control over the tunnel if the number of cars on the island is less than $\hat{n}$, where $\hat{n}$ is a parameter of the system, i.e. it can be fixed arbitrarily. Note that the number of states depends crucially on $\hat{n}$ and that also
the bitwidth of the counters and comparators depend on \( n \). The number of cars on the island (or currently traveling through the tunnel towards the island) is counted by another counter \( IC \). Clearly, the behavior of the entire system depends on the value of \( IC \) in that is is important whether \( IC < n \) holds or not.

More details about the implementation of the island traffic control system can be found in [9]. Here, it is only important to see that the system fulfills all requirements that are necessary for our abstraction. We will demonstrate in the next section how the abstract system is obtained and that this abstract system is even independent from the specific choice of \( n \) such that we can prove the generic system by means of model checking.

### 3.2 The Environment

The design assumes the following assumptions on its environment, described in terms of the behavior of vehicles that entering the system:

**A1:** Cars are not produced in the tunnel, i.e. if there is no car in the tunnel, then no car can exit the tunnel (on none of its sides) at this point of time.

**A2:** Cars do not disappear in the tunnel.

**A3:** Cars are not produced in the island.

**A4:** The island counter counts the cars which are on the island or which are currently inside the tunnel traveling to the island. If all cars on the island are currently inside the tunnel then no car can enter the tunnel at the island side.

**A5:** Each car wants to leave the island after some time.

**A6:** If the island traffic light is green, then cars can only exit the tunnel at the mainland side until the mainland traffic light turns to green (analogously for the mainland).

**A7:** If a car wants to enter a side of the tunnel where the traffic light is green, it will actually enter the tunnel after some time.
The above assumptions can be expressed as follows in temporal logic:

A1: \( G[(TC = 0) \rightarrow \neg il\_exit \land \neg ml\_exit] \)
A2: \( G[(TC \neq 0) \rightarrow F[il\_exit \lor ml\_exit]] \)
A3: \( G(IC = 0) \rightarrow \neg il\_enter \)
A4: \( G(IC = TC) \rightarrow \neg il\_enter \)
A5: \( G(IC \neq 0) \rightarrow F il\_enter \)
A6: \( G[(il\_green \rightarrow [(-il\_exit) \lor ml\_green]) \land (ml\_green \rightarrow [(\neg ml\_exit) \lor il\_green])] \)
A7: \( G[(il\_enter \land il\_green \land X il\_green \rightarrow \neg il\_enter) \land (ml\_enter \land ml\_green \land X ml\_green \rightarrow \neg ml\_enter)] \)

Some of the specifications depend on the fact that only a maximal number of cars is allowed to be on the island or depend on the number of cars in the tunnel, others do not. The latter ones can be proved without the counters and hence, for any number of maximal cars. This implies that some of the following specifications require only a subset of the above assumptions, and some even none of them.

S1: At no point of time, both lights are green.
S2: At no point of time, there are more than \( \hat{n} \) cars on the island.
S3: If a car wants to enter the tunnel and persists to enter the tunnel, then it has the chance to do so after a finite time.
S4: Traffic lights can change only when the tunnel is empty.

The formalizations of the above specifications are as follows:

S1: \( \neg [ml\_green \land il\_green] \)
S2: \( [IC \leq \hat{n}] \)
S3: \( \neg [il\_enter \land il\_red] \land \neg [ml\_enter \land ml\_red] \)
S4: \( G[(ml\_red \land X ml\_green \rightarrow (TC = 0)) \land (il\_red \land X il\_green \rightarrow (TC = 0))] \)

The specification S3 can also be expressed equivalently as follows: If a car wants to enter a side of the tunnel where the light is red and the car waits until the light changes, then the light will actually change after some time. Hence, the formula \( G[\neg [enter \land red]] \) is equivalent to

\[ G[enter \land red \land [(enter \land red) \lor (enter \land green)] \rightarrow F(enter \land green)] \]

### 3.3 The Abstract Verification

The side controllers and the tunnel controller are simple finite state machines with only a small number of states. However, the counters for the number of cars inside the tunnel and the number of cars on the island introduce an arbitrary large number of states depending on the maximal number of cars \( \hat{n} \) that are allowed to be on the island.

We have already stressed that the actual number of cars inside the tunnel is irrelevant. What is important for the decision of the controllers is whether there is a car in the tunnel or not. Hence, we apply our abstraction function with the set \( A_{TC} := \{0\} \) and obtain the abstract finite state machine given in figure [4]. Initially, there are no cars
inside the tunnel, and we remain in this state unless the tunnel counter is incremented or decremented. Then, we switch to corresponding states $TC < 0$ or $TC > 0$. Clearly, the controllers will assure that we will never reach state $TC < 0$, and we have actually verified this. Note that our abstraction leads to a nondeterministic behavior that is modeled by a new oracle input $\zeta$.

The island counter is initialized to 0 and the only comparison is then made with $IC < \hat{n}$. Hence, we apply our abstraction technique with the set $A_{IC} := \{0, \hat{n}\}$ and obtain the abstract finite state machine of figure 5.

We have checked that in the abstract system, both counters will never take negative values and that the island counter will never reach state $IC > \hat{n}$. Also, we have proved that all properties that have been specified for the island traffic control system for the abstract system. The runtimes are given in figure 6 and are measured on a SUN UltraSparc 1 with 192 MBytes main memory, 600 MBytes virtual memory and the model checker SMV version 2.4.3 [10].

The abstract system, viewed as a Kripke structure has only 6912 reachable states, which means that there are only 432 internal states of the circuit. Note that this small finite state machine is a valid abstraction for all instances that are obtained by fixing the maximal number of cars $\hat{n}$.

4 Conclusions

We have presented an abstraction for systems that contain counters to reduce the verification process to small abstract finite state machines that reflect their essential behavior. The abstraction enables us to abstract from systems that use counters where only the
counter’s values are compared with some fixed values. As a result, very small finite state machines are obtained that allow to safely verify all LTL specifications.

The presented case study also shows that our abstraction technique allows even to abstract from generic parameters as e.g. the number of cars in the considered example. Hence, we can prove such systems without using interactive theorem provers. Also, if infinite data types are used for the implementation, our abstraction allows the reduction from the infinite state space to a finite one.

In our future work, we plan to find similar abstraction techniques for other classes than counters which assure that the abstract system simulates the given one. This is important to guarantee that any specification that holds for the abstract system does also hold for the detailed version.

References