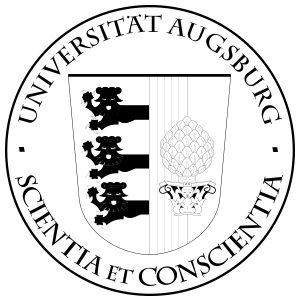
# Universität Augsburg




## Combining Theorem Proving and Model Checking for Verification of Concurrent Systems

**Andriy Dunets, Michael Balser, Wolfgang Reif**

Report 2009-15 August 2009


Institut für Informatik
D-86135 Augsburg

## Abstract

An integration of deductive verification and model checking have been investigated in numerous works over the last decade. We refer to the approaches, where theorem proving was used to reduce verification problems to a form which allows to apply model checking directly [21, 19]. We present a translation procedure from finite state Reactive Logic (RL) [3] specifications of concurrent systems into the SMV model checker. As RL specifications can use arbitrary data types we demonstrate an application of data abstraction using a specification of communication protocol as an example. This paper was motivated by the results achieved in the previous work on verification of medical guidelines by model checking [4]. The basis for this work is an implementation of the symbolic execution proof strategy [1] for concurrent systems in the theorem prover KIV [2].

## 1 Introduction

The growing complexity of distributed and concurrent systems poses a difficult challenge to developers and requires enhanced verification techniques. Automatic verification techniques are very popular in the formal methods community and in the industry as well. However, a well-known limitations to the efficiency of the automatic verification is its applicability only to finite state models and the exponential blow-up of the size of models. Nevertheless, a rather modest amount of human intervention in terms of manually guided abstraction makes possible to analyze industrial strength models very efficiently. Abstractions can be computed in an automatic way or be manually constructed using deductive reasoning.

The importance of deductive verification in computer science is widely recognized. In particular, because of its features like universality, proof reusability, application of common knowledge and possibility to deploy human creativity during the proof process. Still, it can become inefficient proving interactively things which can be more or less easily handled by a machine. In particular, control-oriented properties of concurrent systems can be very efficiently analyzed by model checking.

Model checking is in particular a very efficient technique for the analysis of reactive systems with fairly complex control flows. On the other hand, complex or even unbounded data types in the specification must be previously handled by various abstraction techniques before the model checking algorithm can actually be applied. As data types in theorem proving can be arbitrarily specified we focus on the interactive construction of suitable abstractions.

Our goal is to create an integrated verification environment which combines the automation of model checking with the generality of theorem proving. Establishing of a link between both tools allows to combine the advantages of each of them in different ways and with it increases the efficiency of a verification process.

### 1.1 Related Work

Many techniques have been developed in order to expand the limits of model checking. A construction of property preserving abstractions proved to be an effective way dealing with state explosion or even infinite models. Methods, based on ideas from the framework of *abstract interpretation* [8], have been developed. In particular, the *predicate abstraction* technique [20] automatically constructs (using theorem prover PVS) abstract state graphs based on a set of state predicates $p_1, \ldots, p_n$. [7] presents an algorithm that uses decision procedure to generate finite state abstractions of reactive systems without an exhaustive search of the reachable abstract states (as in PVS method). The advantage of these techniques is that they are fully automatic.

Alternatively, an abstract system can be generated interactively, using theorem proving. In this case, the decrease in the automation is repaid by a flexibility in finding the right abstraction and a richer set of available corrective actions in case a proof fails. In [13] and [10] *data abstraction* is used for the verification of infinite state systems by model checking, while theorem proving is used to establish the correctness of an abstraction. Similar strategy was realized in [19] in the context of Input/Output-Automata (IOA) which represent models of distributed processes. IOAs have been formalized in the theorem prover Isabelle. In this approach safety properties were represented by IOA $\mathcal{A}$ (specification) and the model of a system by IOA $\mathcal{C}$ (implementation). The verification goal of checking $traces(\mathcal{C}) \subseteq traces(\mathcal{A})$ was achieved by introducing IOA $\mathcal{B}$ (abstraction) and checking $traces(\mathcal{C}) \subseteq traces(\mathcal{B})$ (soundness of abstraction) interactively in Isabelle and $traces(\mathcal{B}) \subseteq traces(\mathcal{A})$ by model checking.

This paper is organized as follows. Section 2 introduces the necessary background on the Reactive Logic (RL) in the theorem prover KIV. Section 3 defines a translation procedure from the RL specification of a concurrent system in KIV to the SMV model checker. Section 4 presents an application of data abstraction technique to the producer-consumer example with infinite data types. Section 5 concludes and considers promissing directions for a future work.

## 2 Reactive Logic in KIV

KIV [2] is an integrated environment for system development using formal methods. KIV supports both the functional and the state-based approaches to specify systems. Higher-order algebraic specifications of software and systems designs are represented as directed acyclic graphs, called *development graphs*. Properties of a design are verified by an interactive construction of proofs. KIV combines a high degree of automation with an elaborate interactive proof engineering environment. Support for the development of concurrent systems has recently been added.

The Reactive Logic (RL) [3] was defined and implemented in KIV for the purpose of deductive reasoning about and verification of concurrent systems. The RL

syntax and the semantics of programs is close to SPL [22], however, it is more in the style of the Interval Temporal Logic [18], where programs and formulas can be mixed. In RL the behavior of concurrent systems is described using parallel programs and their properties are formulated in Linear Temporal Logic (LTL). A state of a system is encoded in first-order logic using variables. We distinguish between *static* variables $v$ (represented using lowercase), which do not change their value over time, and *dynamic* variables $V$. The most distinctive attribute of this logic are double-primed variables. A primed variable $V'$ represents the value of $V$ after a system transition, the double-primed variable $V''$ stores the value after an environment transition. System and environment transition alternate, while $V''$ is equal to the value of $V$ in the successive state. This feature is introduced in order to get a compositional semantics for interleaving operator $||_i$.

We refer to [3] for a detailed definition of RL syntax and semantics.

## 3 Generating Finite-State Model

Usually finite state transition systems are represented using *Kripke models*. In the context of concurrent programs, in order to construct a Kripke model for a given concurrent system, we should provide the set of system variables $\mathcal{V}$ as well as the first order formulas $\mathcal{I}$ and $\mathcal{T}$ specifying the set of initial states and the transition relation respectively.

$$\varphi_{init},\ [\alpha],\ \psi_{env}\ \vdash\ \chi$$

**Fig. 1.** Proof obligation in KIV: *precondition, program, environment assumption* $\vdash$ *temporal property*

Consider a temporal proof obligation in the theorem prover KIV, see Fig. 1. In RL programs and formulas can be mixed as well as complex abstract data types can be used. Therefore, we must put some restrictions on the precondition $\varphi_{init}$ and the program $[\alpha]$, see Definition 1, in order to be able to construct the corresponding finite-state model directly (i.e. without any use of abstraction) and automatically from the underlying specification specification in KIV. We intentionally abandon the option of using well-established abstractions for program analysis at this point, e.g. the method of *predicate abstraction* which uses decision procedures for the automatic construction of an abstraction for a given program. We are investigating an option of an interactive way of an *abstraction* construction in the the theorem prover KIV using deductive reasoning. The theorem prover is used to prove a homomorphic correspondence between the concrete and the abstract model. Therefore, we assume that a model of a concurrent system is already finite, i.e. satisfies certain assumptions.

**Definition 1 (Assumptions).** *We assume that:*

- *all program variables have finite domain and are correspondingly initialized in the precondition $\varphi_{init}$, e.g. $\varphi_{init} \equiv N = 0 \wedge M = 1$*
- *programs contain no recursive procedure calls*
- *as $||_s$,$||_i$ are associative operators, we assume that program $P = P_1 || \ldots || P_n$ (flat hierarchy)*

Given a proof obligation, see Fig. 1, which fulfils the assumptions from the Definition 1. In order to find out by model checking whether the concurrent program $[\alpha]$ satisfies the LTL property $\chi$ under the environment assumption $\psi_{env}$, we should construct a corresponding finite-state model. This finite-state model is specified by providing a set of system variables $\mathcal{V}$ (which implicitly defines the state space) as well as first order formulas $\mathcal{I}$ and $\mathcal{T}$ (initial states and transition relation). The formula $\mathcal{I}$ can be straightforward derived from the precondition $\varphi_{PL}$. The details on construction of $\mathcal{V}$ and $\mathcal{T}$ are described in 3.1 and 3.2. In case of RL, the formula $\mathcal{T}$ is defined as a composition of two relations $\mathcal{T}_s$ and $\mathcal{T}_e$ which model the transitions of a concurrent program and of an environment respectively, i.e. $\mathcal{T} := \mathcal{T}_s \ ; \ \mathcal{T}_e$. This corresponds to the semantics of RL, where the TL step is defined as a successive combination of the step of a system (modelled by a concurrent program) followed by the step of an environment (modelled by an environment assumption). Next we will describe an approach to a construction of $\mathcal{T}_s$ (for a given program $[\alpha]$) and of $\mathcal{T}_e$ (for a given environment assumption expressed by an LTL formula $\psi_{env}$).

### 3.1 Programs

For the construction of $\mathcal{T}_s$ we use the approach presented in the books by Manna and Pnuelli [16] and by Clarke, Grumberg and Peled [6] (Chapter 2), which is based on the idea of labelling of statements in a program. A translation procedure $\mathcal{C}$ is defined, that takes the text of a labeled program $P$ and transforms it into first order formula $\mathcal{T}_s$, where $\mathcal{T}_s$ represents the set of transitions of the program. Further, the definition of $\mathcal{C}$ is extended for the interleaved operator $||_i$. Also, a labelling transformation of a program $P$ into a labeled program $P^{\mathcal{L}}$ is defined.

The definition of a labeling transformation of a program is based on the assumption that each statement in a program has a unique *entry point* and a unique *exit point*. As in sequential programs the exit point of a statement is equal to the entry point of the following statement, it is sufficient for the labeling transformation to attach labels only for the entry points of statements of a program $P$. Further, labels for the entry and exit points of $P$ must be provided. We assume that no two attached labels are identical. For an arbitrary program $P$ the labeling transformation to $P^{\mathcal{L}}$ is defined as follows, see Definition 2:

**Definition 2 (labeled program $P^{\mathcal{L}}$).** *Consider the following cases:*

- *if $P$ is a basic statement (e.g. $P$ is $x := e$, **skip**, **await**), then $P^{\mathcal{L}} = P$.*

- *if* $P = P_1 \,;\, P_2$, *then* $P^{\mathcal{L}} = P_1^{\mathcal{L}} \,;\, l : P_2^{\mathcal{L}}$.
- *if* $P = \textbf{\textit{if}}\ c\ \textbf{\textit{then}}\ P_1\ \textbf{\textit{else}}\ P_2$, *then* $P^{\mathcal{L}} = \textbf{\textit{if}}\ c\ \textbf{\textit{then}}\ l_1 : P_1^{\mathcal{L}}\ \textbf{\textit{else}}\ l_2 : P_2^{\mathcal{L}}$.
- *if* $P = \textbf{\textit{while}}\ c\ \textbf{\textit{do}}\ P_1$, *then* $P^{\mathcal{L}} = \textbf{\textit{while}}\ c\ \textbf{\textit{do}}\ l_1 : P_1^{\mathcal{L}}$.
- *if* $P = P_1||P_2||\ldots||P_n$, *then* $P^{\mathcal{L}} = l_1 : P_1^{\mathcal{L}}||l_2 : P_2^{\mathcal{L}}||\ldots||l_n : P_n^{\mathcal{L}}$.

In following, we assume that $P$ as well as all $P_i$ are already labeled programs with entry and exit points labeled $l, l'$ and $l_i, l_i'$ respectively. For each process $P_i$ we introduce a special variable $pc_i$ called *program counter* that ranges over the set of program labels.

The translation procedure $\mathcal{C}$ has three parameter: the entry label $l$, the labeled program $P$ and the exit label $l'$. For a given concurrent program $P$, the corresponding transition relation $\mathcal{T}_s$ is defined as $\mathcal{C}(l, P, l')$, where $\mathcal{C}(l, P, l')$ is a disjunction of all possible transitions. A single transition is represented by a conjunction which is true whenever the transition is enabled and false otherwise. $\mathcal{C}$ is recursively defined for each program construct, see Definition 3.

**Definition 3 (translation procedure $\mathcal{C}$).** *In order to define* $\mathcal{C}(l, P, l')$ *consider the following cases for* $P$*:*

- $\mathcal{C}(l, x := e, l') \equiv pc = l \ \wedge\ pc' = l' \ \wedge\ x' = e \ \wedge\ same(V \backslash \{x, pc\})$ [1]
- $\mathcal{C}(l, \textbf{\textit{skip}}, l') \equiv pc = l \ \wedge\ pc' = l' \ \wedge\ same(V \backslash \{pc\})$
- $\mathcal{C}(l, \textbf{\textit{await}}\ c, l') \equiv (pc = l \ \wedge\ pc' = l' \ \wedge\ c \wedge\ same(V \backslash \{pc\})) \ \vee$
  $(pc = l \ \wedge\ pc' = l \ \wedge\ \neg c \wedge\ same(V))$
- $\mathcal{C}(l, P_1 \,;\, l'' : P_2, l') \equiv \mathcal{C}(l, P_1, l'') \ \vee\ \mathcal{C}(l'', P_2, l')$
- $\mathcal{C}(l, \textbf{\textit{if}}\ c\ \textbf{\textit{then}}\ l_1 : P_1^{\mathcal{L}}\ \textbf{\textit{else}}\ l_2 : P_2^{\mathcal{L}}, l') \equiv$
  $(pc = l \ \wedge\ pc' = l_1 \ \wedge\ c \wedge\ same(V \backslash \{pc\})) \ \vee$
  $(pc = l \ \wedge\ pc' = l_2 \ \wedge\ \neg c \wedge\ same(V \backslash \{pc\})) \ \vee\ \mathcal{C}(l_1, P_1, l') \ \vee\ \mathcal{C}(l_2, P_2, l')$
- $\mathcal{C}(l, \textbf{\textit{while}}\ c\ \textbf{\textit{do}}\ l_1 : P_1^{\mathcal{L}}, l') \equiv$
  $(pc = l \ \wedge\ pc' = l_1 \ \wedge\ c \wedge\ same(V \backslash \{pc\})) \ \vee$
  $(pc = l \ \wedge\ pc' = l' \ \wedge\ \neg c \wedge\ same(V \backslash \{pc\})) \ \vee\ \mathcal{C}(l_1, P_1, l)$
- $\mathcal{C}(l, l_1 : P_1^{\mathcal{L}}||_i l_2 : P_2^{\mathcal{L}}||_i \ldots||_i l_n : P_n^{\mathcal{L}}, l') \equiv$
  $fairSched \ \wedge\ \bigvee_{i=1}^{n}(\neg blk_i \ \wedge\ \mathcal{C}(l_i, P_i, l_i'))$
- $\mathcal{C}(l, l_1 : P_1^{\mathcal{L}}||_s l_2 : P_2^{\mathcal{L}}||_i \ldots||_s l_n : P_n^{\mathcal{L}}, l') \equiv$
  $\bigwedge_{i=1}^{n} \mathcal{C}(l_i, P_i, l_i')$

In contrast to [16] and [6], the translation procedure $\mathcal{C}$ in Def. 3 treats asynchronous parallel operator $||_i$ in a different way, i.e. a process is scheduled only if it is not blocked ($\neg blk_i$ holds) [2]. Also, the interleaved operator $||_i$ in RL *weakly fair*, i.e. if a process is infinitely often not blocked, eventually it will be scheduled.

[1] $same(V) \iff \forall\, x \in V.\ x' = x$

[2] A process is blocked *iff* it stalls at a synchronization point *await* $c$, i.e. the condition $c$ is currently *false*

### 3.2 Environment

The most commonly used environment assumption is one where the environment is assumed to be not modifying the system variables, e.g. $\psi_{env} \equiv \Box N'' = N'$. Another kind of frequently used assumptions are liveness properties which state that an event eventually occurs, e.g. $\Diamond Push'' \leftrightarrow true$.

As we mentioned before, the purpose of introducing double primed variables in RL is to make the logic more suitable for the compositional reasoning about concurrent systems. It is possible, for example, to consider separately only one component of a concurrent system for the verification, while replacing the other parts by their abstractions in form of LTL properties. These LTL properties are put in the environmet assumptions during the verification.

Therefore, in general, we can get an arbitrary LTL formula $\psi_{env}$ describing an environment behavior, i.e. a relationship between double primed variables $\nu''$ and primed variables $\nu'$. For example, $\psi_{env}$ can contain propositions $p(\nu'') \in \mathcal{L}(\psi_{env})$ (here $\mathcal{L}(\psi_{env})$ represents the set of propositions which occur in $\psi_{env}$) which have form $\nu'' = \tau(\nu'_1, .., \nu'_n)$, where $\nu$ and $\nu_i$ represent program variables and $\tau$ is a function constructed using operations like addition, subtraction etc.

In this Section we describe how given an environment assumption $\psi_{env}$ a corresponding environment transition relation can be constructed. We use an algorithm of Gerth, Peled, Vardi and Wolper [11] for translating an LTL path formula into a generalized Büchi automaton using depth-first search. This algorithm is designed to produce small automata and avoids the exponential blowup that can occur in their construction whenever possible. Further, a simple translation from a generalized Büchi automaton to a Büchi automaton is performed. After the Büchi automaton, see Definition 4, for a given LTL formula $\psi_{env}$ is constructed, we have to translate it into the first order formula $\mathcal{T}_e$ representing the corresponding transition relation.

**Definition 4 (Büchi automaton).** $\mathcal{A} = \langle Q, I, \rightarrow, F \rangle$, *where:*

- $Q$ *is a finite set of states*
- $I \subseteq Q$ *is the set of initial states*
- $\rightarrow \subseteq Q \times Q$ *is the transition relation*
- $F \subseteq 2^Q$ *is the set of accepting states.*

*An execution* $\mathcal{A}$ *is an infinite sequence* $\sigma = q_0 q_1 q_2 \ldots$ *such that* $q_0 \in I$ *and* $\forall i \geq 0. g_i \rightarrow q_{i+1}$. *An accepting execution* $\sigma$ *is an execution such that,* $\forall q \in F.\ q \in inf(\sigma)$ [3].

Consider the LTL formula $\Box(p \rightarrow (rUq))$[4] as an example. The corresponding Büchi automaton is shown on Fig. 2. In order to translate this automaton into $\mathcal{T}_e$ we introduce an additional state variable $s$ ranging over $\{s_0, s_1\}$:

[3] $q$ appears infinitely often in the execution $\sigma$

[4] as mentioned above, $p, q, r$ abbreviate some relationships between primed and double primed variables, e.g. $N'' = N'$

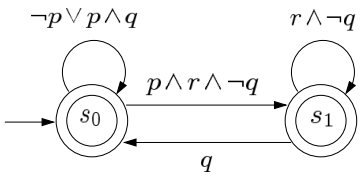



**Fig. 2.** Büchi automaton for $\Box(p \rightarrow (rUq))$

$$
\begin{aligned}
&(s = s_0 \ \wedge \ s' = s_0 \ \wedge \ (\neg p \ \vee \ p \ \wedge \ q) \ \vee \\
&s = s_0 \ \wedge \ s' = s_1 \ \wedge \ p \ \wedge \ r \ \wedge \ \neg q \ \vee \\
&s = s_1 \ \wedge \ s' = s_1 \ \wedge \ r \ \wedge \ \neg q \ \vee \\
&s = s_1 \ \wedge \ s' = s_0 \ \wedge \ q) \ \wedge \ fair(s = s_0 \vee s = s_1)
\end{aligned}
$$

**Fig. 3.** Formula representing Büchi automaton on Fig. 2

The assumption $fair(s = s_0 \vee s = s_1)$ requires that $s = s_0 \vee s = s_1$ holds infinitely often on every trace. This fairness constraint models the acceptance condition for the Büchi automaton on Fig. 2.

### 3.3 Encoding into SMV

As we previously used SMV [14] model checker (Cadence SMV) for the verification of medical guidelines [4] and it proved to be a very efficient tool, we also decided to use it for the model checking of RL programs. In particular, it provides such useful for our purposes constructs like *TRANS* and *INVAR*. The first one allows to encode a transition relation of state variables by a formula, e.g. see Fig. 3. The *INVAR* statement restricts the state space only to states satisfying a given condition. In order to model the weak fairness of the interleaved operator $||_i$ the *FAIRNESS* construct of SMV language is used. This construct allows to restrict a model only to traces where the declared constraint holds infinitely often. For example, by declaring *FAIRNESS blk1 | arbiter=1* we exclude those possible traces where the first process is infinitely often (in the weak sense) not blocked and still is not scheduled.

Consider, as an example, the Fisher's mutual exclusion protocol modelled by concurrent programs, see Fig. 4.

According to the translation schema presented in 3.1 and 3.2 the formulas $\mathcal{T}_s$ and $\mathcal{T}_e$[5] modelling the system step and the environment step respectively are

[5] the set of accepting states of the corresponding Büchi automaton is empty

$$
\begin{array}{lll}
 & P = 1,\ Q = 0,\ R = 0, & (:\ \texttt{initial values} \quad :) \\
 & [\textbf{mutex}], & (:\ \texttt{parallel program} \quad :) \\
 & \Box\ (P'' = P'\ \wedge\ Q'' = Q'\ \wedge\ R'' = R') & (:\ \texttt{environment assumption} :) \\
\vdash & \Box\ (Q \neq 1\ \vee\ R \neq 1) & (:\ \texttt{property to prove} \quad :)
\end{array}
$$

where

**mutex** $\equiv$

| | |
|---|---|
| **while** *true* **do begin** | **while** *true* **do begin** |
| **await** $P = 1$; | **await** $P = 0$; |
| (: critical section :) | (: critical section :) |
| $Q := 1$; | $R := 1$; |
| (: noncritical section :) | (: noncritical section :) |
| $Q := 0,\ P := 0$; | $R := 0,\ P := 1$; |
| **end** | **end** |

$_i$

**Fig. 4.** Proof obligation in KIV for the Fisher's mutual exclusion protocol safety property.

automatically generated for this example:

$$
\begin{aligned}
\mathcal{T}_s \equiv\ & (\neg blk_1\ \wedge\ ((pc_1 = l_{11}\ \wedge\ pc_1' = l_{12}\ \wedge\ true\ \wedge\ same(\{P,Q,R\}))\ \vee \\
& (pc_1 = l_{11}\ \wedge\ pc_1' = l_{15}\ \wedge\ \neg\, true\ \wedge\ same(\{P,Q,R\}))\ \vee \\
& (pc_1 = l_{12}\ \wedge\ pc_1' = l_{13}\ \wedge\ P = 1\ \wedge\ same(\{P,Q,R\}))\ \vee \\
& (pc_1 = l_{12}\ \wedge\ pc_1' = l_{12}\ \wedge\ \neg\,(P = 1)\ \wedge\ same(\{P,Q,R\}))\ \vee \\
& (pc_1 = l_{13}\ \wedge\ pc_1' = l_{14}\ \wedge\ Q' = 1\ \wedge\ same(\{P,R\}))\ \vee \\
& (pc_1 = l_{14}\ \wedge\ pc_1' = l_{11}\ \wedge\ Q' = 0\ \wedge\ P' = 0\ \wedge\ same(\{R\}))\ \vee \\
& (pc_1 = l_{15}\ \wedge\ pc_1' = l_{15}\ \wedge\ same(\{P,Q,R\}))\ \wedge\ same(\{pc_2\}))\ \vee \\
& (\neg blk_2\ \wedge\ (\ldots)\ \wedge\ same(\{pc_1\})), \\
& \quad where\ blk_1 \equiv pc_1 = l_{12}\ \wedge\ \neg(P = 1),\ blk_2 \equiv pc_2 = l_{22}\ \wedge\ \neg(P = 0) \\
\mathcal{T}_e \equiv\ & s = s_0\ \wedge\ s' = s_0\ \wedge\ P'' = P'\ \wedge\ Q'' = Q'\ \wedge\ R'' = R'
\end{aligned}
$$

After the translation of the program $[mutex]$ and the environment assumption $\Box(P'' = P'\ \wedge\ \ldots)$ into formulas the corresponding SMV model is generated. For reasons of better readability of the generated SMV model we encode the transition relation $\mathcal{T}_s$ indirectly using *case* construct. On the other hand, the transition relation $\mathcal{T}_e$ of an environment is directly encoded into SMV using $TRANS$ construct.

In RL a temporal step consists of two substeps (system step and environment step) which are executed sequentially, i.e. first the value of primed variables $\nu'$ is computed and then the value of double primed variables depending on the

```
module main(blk1,blk2,Pp,Qp,Rp) {
p : {1,0};  output pp : {1,0};  q : {1,0};  output qp : {1,0};
r : {1,0};  output rp : {1,0};
init(p) := 1;  init(q) := 0;  init(r) := 0;

arbiter : {ini,no,1,2};
pc1 : {l1x1,l1x2,l1x3,l1x4,l1x5};  pc2 : {l2x1,l2x2,l2x3,l2x4,l2x5};
init(pc1) := l1x1;  init(pc2) := l2x1;
init(arbiter) := ini;  next(arbiter) := {no,1,2};

output blk1 : {0,1};  blk1 := pc1=l1x2 & !p=1 | pc1=l1x5;
output blk2 : {0,1};  blk2 := pc2=l2x2 & !!p=1 | pc2=l2x5;
case {
   arbiter=1 : case   {
      pc1=l1x1 & 1 : next(pc1) := l1x2;
      pc1=l1x1 & !(1) : next(pc1) := l1x5;
      pc1=l1x2 & p=1 : next(pc1) := l1x3;
      pc1=l1x2 & !(p=1) : next(pc1) := pc1;
      pc1=l1x3 : next(pc1) := l1x4;
      pc1=l1x4 : next(pc1) := l1x1;
      1 : next(pc1) := pc1;
   }
   1 : next(pc1) := pc1;
}
... /* case for arbiter=2 : transition relation of pc2 */
case {
   arbiter=1 : case   {
      pc1=l1x4 : pp := 0;
      1 : pp := p;
   }
   arbiter=2 : case
   {
      pc2=l2x4 : pp := 1;
      1 : pp := p;
   }
   1 : pp := p;
}
... /* definitions of Qp and Rp */
INVAR
!(blk1=1 & arbiter=1) & !(blk2=1 & arbiter=2) &
!((blk1=0 | blk2=0) & arbiter=no);

s : {1}; init(s) := {1};
TRANS  ((s=1) & next(s)=1 & next(r)=rp & next(q)=qp & next(p)=pp);

FAIRNESS (blk1 | arbiter=1) & (blk2 | arbiter=2);

assert G !(q=1 & r=1);
}
```

**Fig. 5.** The SMV model excerpt for the mutex protocol

assignment of primed variables. We modelled this behavior by introducing for each variable $v$ an output parameter *output* : $v_p$ and a state variable $v$. The value of $v_p$ in each step step represents the value of the primed variable $v'$ and is computed from the current values of the state variables in an unambiguous way due to the definition of $\mathcal{T}_s$. The value of $v''$ is defined by *next(v)* (the value of the state variable $v$ in the next step) due to the definition of $\mathcal{T}_e$.

## 4 Data Abstraction

Usually, RL specifications contain complex and infinite data types. Unlike theorem proving, which is very suitable for reasoning about abstract data types, model checking puts restrictions on the used data types, e.g., they should be finite. Consequently, the first step in an application of model checking to the specifications of concurrent systems in a theorem prover is to transform them to a finite state form. This step typically involves some kind of abstraction. As we focus on the abstraction of the data part of a model (model checking is quite efficient in handling control part) we will use *data abstraction* [12, 5, 9, 13] technique.

The soundness of verification techniques involving abstractions essentially depends on the property preservation. Whenever the abstract system satisfies a formula, the concrete system also satisfies this formula. In our framework we are dealing with LTL formulas. Since LTL formulas are interpreted over all possible traces of a system (the system satisfies LTL formula $\Leftrightarrow$ LTL formula is satisfied on every trace of the system), it is sufficient to show that for every trace of the concrete system $M$ there exists a corresponding trace in the abstract system $M'$. This correspondence is encoded as a relation between data types in the concrete system and their correspondents in the abstract system. These ideas are formalized by the notion of *simulation relation* [17]. Assuming that $M'$ *simulates* $M$, then according to [5] $M'$ preserves ACTL (therewith LTL) properties of $M$.

As in our case $M$ is defined by a parallel program operating on variables, we define $M'$ by providing a surjective *abstraction mapping* $\alpha$ from concrete to abstract domains. Usually, it is defined as an identity mapping for finite domains and an equivalence relation for infinite domains respectively. Further, for each function $f_c$ in the concrete program an abstract counterpart $f_a$ should be provided. Abstract functions $f_a$ introduce nondeterminism which is modeled by nondeterministic operations, i.e. set-valued functions. In this context, our original requirement on $M'$ to "mimic" $M$ can be formulated by using a notion of *homomorphism*, see Def. 5. In fact, if $\alpha$ is proven to be an homomorphism with respect to the concrete functions $f_c$ then it defines a simulation relation between $M$ and $M'$ and therefore is a safe abstraction [15].

**Definition 5 (Homomorphic mapping).** *An abstraction mapping $\alpha$ is an homomorphism with respect to functions $f_c$ and their abstract counterparts $f_a$ if*

*following holds:*

$$\forall \bar{x}.\ \alpha(f_c(\bar{x}))\ \in\ f_a(\alpha(\bar{x}))$$

In following, we will demonstrate an application of data abstraction followed by a formal proof of simulation relation in the theorem prover KIV using a concrete example.

### 4.1 Example: Producer-Consumer

Consider an asynchronous concurrent system consisting of two processes: *producer*, which generates random values, and *consumer*, which receives these values, see Fig. 6. The communication is abstracted by a shared variable $CH.data$ and works according to the handshake procedure: the receiver is informed by the signal $CH.sig$ that something is sent and acknowledges a reception by setting the signal $CH.ack$ (a signal is sent by negating the corresponding variable). Furthermore, both processes store the messages which they have sent/received in the queues *PLIST/CLIST*. Initially, the queues are empty, channel contains some random data and both signals are set to true. The property we want to prove argues that no messages are lost, no doubles are produced by the channel and the order of messages is preserved. It is formulated as a safety property: the consumer queue $CLIST$ is always a prefix of the producer queue $PLIST$.

$$\begin{array}{ll} & PLIST = [\,],\ CLIST = [\,],\ CH = mkch(A, true, true), \\ & [\textbf{prodcon}], \\ & \Box\ (CH'' = CH'\ \wedge\ A'' = A'\ \wedge\ PLIST'' = PLIST'\ \wedge\ CLIST'' = CLIST') \\ \vdash & \Box\ (CLIST \subseteq PLIST) \end{array}$$

where

**prodcon** $\equiv$

| | | |
|---|---|---|
| **while** *true* **do begin** | $\|\|_i$ | **while** *true* **do begin** |
| **await** $CH.sig = CH.ack$; | | **await** $CH.sig \neq CH.ack$; |
| $A := [?]$; | | $A := CH.data$; |
| $PLIST := PLIST + A$; | | $CLIST := CLIST + A$; |
| $CH := mkch(A, \neg CH.sig, CH.ack)$; | | $CH := mkch(CH.data, CH.sig, \neg CH.ack)$; |
| **end** | | **end** |

**Fig. 6.** Proof obligation in KIV for the Producer-Consumer example.

Although, this example has a quite simple control structure, it can not be directly model checked as its specification contains two infinite data types: *elem* (unspecified message type) and *list* (lists of elements with arbitrary lengths). The abstraction, we will use, is motivated by following observations done by an informal analysis of the system:

- property actually does not care about a concrete value of the variable $A$ as long as it is delivered on time
- in every step of an execution the lengths of both queues are either equal or $\#MB.pli = 1 + \#MB.cli$.

The RL specification of Producer-Consumer example uses following sorts: *bool, elem, list* and *channel*. Here, sorts *bool* (finite) and *channel* (structure) pose no problem with respect to finiteness, while sorts *elem* and *list* should be treated by an abstraction. Instead of defining abstract sorts $elem_a$ and $list_a$ followed by a definition of mappings $\alpha : elem \rightarrow elem_a$ and $\alpha : list \rightarrow list_a$ we follow a bit different way of defining a data abstraction. First, we transform the specification by grouping all variables with *infinite* domain in a tuple, e.g. variables *PLIST, CLIST* and $A$ are bundled in *[PLIST,CLIST,A]*, and define a new data type $Tuple$ which is a structure where each place corresponds to some infinite variable in program[6]. Next, we introduce a new program variable $T : Tuple$ and replace all occurrences of bundled variables $v$ in a program by $T.v$. Now we will define an abstraction mapping $\alpha$ for data type $Tuple$. The corresponding abstract data type $Tuple_a$ has a finite domain which contains symbolic values. The variable $T$ in the concrete program is mapped to $T_a : Tuple_a$ in the abstract program. All assignments $T.v := \tau$ in the concrete program are mapped on assignments $T_a := \alpha(\tau)$ in the abstract program. For each function $f_c$ in the concrete program which is applied to $T.v$, i.e. $f_c$ is applied to $T$, we define an abstract function $f_a$ which is applied just to $T_a$, e.g. $PLIST := PLIST + A$ is transformed into $T.PLIST := T.PLIST + T.A$ and mapped on $T_a := pinsert_a(T_a)$ in the abstract program. Special case here is the nondeterministic assignment $A := [?]$ to variable $A$, which is mapped on the abstract function $getrandomA : Tuple_a \rightarrow Tuple_a$. The overall definition of the abstraction mapping $\alpha$ for the Producer-Consumer example is shown in Fig. 7.

As the data type *channel*, which describes a structure, has infinite data type *elem* as a component, it is mapped by $\alpha$ not exactly on itself (as *bool* for example). Abstract version of *channel* has component of type $Tuple_a$ instead of *elem*. According to the definition of the mapping $\alpha$ an abstract program is constructed, see Fig. 8.

The temporal property $\Box PLIST \sqsubseteq CLIST$ can be rewritten as $\Box\ \exists z.\ PLIST = CLIST + z$. Obviously our abstraction is too coarse in order to represent the $\sqsubseteq$ function in an appropriate way. So we show a stronger temporal property

[6] Surely, infinite variables in a program can be grouped in several disjoint tuples depending on an abstraction which is used.

$$
\begin{aligned}
\alpha(mkTuple(x,y,A)) &= \begin{cases} Equal & \text{if } x = y \\ Shorter1 & \text{if } x = y + A \\ \bot & \text{otherwise} \end{cases} \\
pinsert_a(T_a) &= \begin{cases} \{Shorter1\} & \text{if } T_a = Equal \\ \{Equal, \bot\} & \text{otherwise} \end{cases} \\
cinsert_a(T_a) &= \begin{cases} \{Equal\} & \text{if } T_a = Shorter1 \\ \{Shorter1, \bot\} & \text{otherwise} \end{cases} \\
getrandomA(T_a) &= \begin{cases} \{T_a\} & \text{if } T_a = Equal \\ \{\bot, Shorter1\} & \text{otherwise} \end{cases}
\end{aligned}
$$

**Fig. 7.** Definition of the abstraction mapping $\alpha$ and of the corresponding abstract functions

$\Box\ PLIST = CLIST \ \vee\ PLIST = CLIST + A$ which can be directly mapped on $\Box\ T_a = Equal \ \vee\ T_a = Shorter1$.

$$
\begin{aligned}
& T_a = Equal,\ CH_a = mkch(T_a, true, true), \\
& [\textbf{prodcon}], \\
& \Box\ (CH'' = CH' \ \wedge\ T_a'' = T_a') \\
\vdash\ & \Box\ (T_a = Equal \vee T_a = Shorter1)
\end{aligned}
$$

where

**prodcon** $\equiv$

| | |
|---|---|
| **while** *true* **do begin** | **while** *true* **do begin** |
| **await** $CH.sig = CH.ack$; | **await** $CH.sig \neq CH.ack$; |
| $T_a := getrandomA(T_a)$; | $T_a := CH_a.data$; |
| $T_a := pinsert_a(T_a)$; | $T_a := cinsert_a(T_a)$; |
| $CH_a := mkch(T_a, \neg CH.sig, CH.ack)$; | $CH_a := mkch(CH_a.data, CH.sig, \neg CH.ack)$; |
| **end** | **end** |

$_i$

**Fig. 8.** Abstract finite state program and corresponding proof obligation.

In order to assure that an abstraction mapping $\alpha$ is safe with respect to LTL properties we have to prove that it is an homomorphism with respect to used functions $f$ in a concrete program and their abstractions $f_a$ in the abstract program, see Def. 5. For example, for function $pinsert : Tuple \rightarrow Tuple$ (corresponds to $PLIST := PLIST + A$) we have to show that $\forall T.\ h(pinsert(T)) \in pinsert_a(h(T))$. All these proofs were automatically accomplished in KIV using the standard set of heuristics (in particular automatic case splitting as abstract

functions usually represent big case distinctions). In the first try, we defined $pinsert_a(T_a)$ for the case $T_a \neq Equal$ to be equal $\{\bot\}$. By proving the homomorphism property for the abstract function $pinsert_a$ we discovered an error : by inserting an element in $PLIST$ we can make this list equal $CLIST$. Therefore, the right definition of $pinsert_a(T_a)$ for the case $T_a \neq Equal$ is $\{\bot, Equal\}$, see Def. 7.

After an abstract finite state program is constructed, we can use the automatic method from Section 3 and start the SMV model checker. Automatic proof of the abstract property $\Box\ (T_a = Equal \vee T_a = Shorter1)$ takes virtually no time (2000 BDD nodes were allocated) and the property is proven to hold for the abstract model, which implies that original property $\Box\ CLIST \sqsubseteq PLIST$ holds in the original model as well. Practical experiences from the verification of Producer-Consumer example have shown that the most effort was put in finding an idea of an abstraction $\alpha$ as well as defining abstract functions $pinsert_a, cinsert_a$. Following table gives an overview of model checking results (all experiments were carried out on a AMD Athlon Dual Core with two 2.4 GHz processors and 4GB RAM memory):

| Example | code lines | allocated BDD nodes | SMV time |
|---|---|---|---|
| mutex (generated SMV model) | 83 | 634 | 0.01s |
| mutex (classical SMV model) | 43 | 421 | 0.01s |
| prodcon | 125 | 2256 | 0.01s |

## 5 Conclusion and Future Work

We have presented a technique which combines model checking and theorem proving within the Reactive Logic framework in the theorem prover KIV. It allows to prove arbitrary linear time temporal logic properties of concurrent systems. We define a translation procedure from finite state RL specifications into finite state transition systems which are subsequently encoded into SMV syntax. Here, we extended the classical translation procedure for parallel programs [16, 6] to RL which supports the concept of an environment.

As typical RL specifications contain arbitrary infinite data types, e.g. infinite lists, we demonstrated an application of the well-known data abstraction technique, which allows to bring specifications to a finite form. This kind of abstraction is an informal activity which requires a thorough understanding of a model, i.e. it requires a reasonable human interaction effort. Though, all generated proof obligations are automatically discharged in the theorem prover KIV, in case a proof fails, human interaction is required in order to adjust the abstraction mapping. The proof of a trace inclusion does not prevent an abstract model from containing spurious counter examples. In fact, there exists a trade-off between defining as lazy abstraction as possible (which results in nondeterminism and spurious counter examples) and defining a very precise abstraction (state explo-

sion). Therefore, it would be very helpful to enable an automatic simulation of generated counter examples on the concrete model, in order to detect spurious ones.

By introducing double primed variables, which enables to replace some parts of a model by an environment assumption, Reactive Logic is very suitable for compositional reasoning. It looks like very promising to apply our method to larger practical examples in combination with compositional strategies.

## References


1. M. Balser, C. Duelli, W. Reif, and G. Schellhorn. Verifying concurrent systems with symbolic execution. *Journal of Logic and Computation*, 12(4):549–560, 2002.
2. M. Balser, W. Reif, G. Schellhorn, K. Stenzel, and A. Thums. Formal system development with KIV. In T. Maibaum, editor, *Fundamental Approaches to Software Engineering*, number 1783 in LNCS. Springer, 2000.
3. Michael Balser. *Verifying Concurrent Systems with Symbolic Execution*. Shaker-Verlag, 2006.
4. S. Bäumler, M. Balser, A. Dunets, W. Reif, and J. Schmitt. Verification of medical guidelines by model checking - a case study. In *SPIN*, volume 3925 of *Lecture Notes in Computer Science*, pages 219–233. Springer, 2006.
5. E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, September 1994.
6. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 2000.
7. M. Colón and T. Uribe. Generating finite-state abstractions of reactive systems using decision procedures. In *Proceedings of the Conference on ComputerAided Verification*, number 1427 in Lecture Notes in Computer Science, pages 293–304. Springer-Verlag, July 1998.
8. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.
9. D. E. Long. *Model Checking, Abstraction and Compositional Verification. PhD thesis. School of Computer Science, Carnegie Mellon University*. July 1993.
10. Dennis Dams, Dieter Hutter, and Natalia Sidorova. Combining theorem proving and model checking — a case study, 2000.
11. Rob Gerth, Doron Peled, Moshe Y. Vardi, and Pierre Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Protocol Specification Testing and Verification*, pages 3–18, Warsaw, Poland, 1995. Chapman & Hall.
12. O. Grumberg. Abstractions and reductions in model checking. In *Marktoberdorf summer school*. Nato Science Series, 2001.
13. J. Dingel and T. Filkorn. Model checking for infinite state systems using data abstraction, assumption-commitment style reasoning and theorem proving. In P. Wolper, editor, *Proceedings of the 7th International Conference On Computer Aided Verification*, volume 939, pages 54–69, Liege, Belgium, 1995. Springer Verlag.
14. K.L. McMillan. The SMV system. Technical Report CMU-CS-92-131, 1992.

15. Claire Loiseaux, Susanne Graf, Joseph Sifakis, Ahmed Bouajjani, and Saddek Bensalem. Property preserving abstractions for the verification of concurrent systems. *Formal Methods in System Design*, 6(1):11–44, 1995.
16. Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems - Safety.* Springer-Verlag, 1995.
17. Robin Milner. An algebraic definition of simulation between programs. Technical report, Stanford, CA, USA, 1971.
18. B. Moszkowski. A temporal logic for multilevel reasoning about hardware. IEEE Computer 18(2), pages 10–19, 1985.
19. Olaf Muller and Tobias Nipkow. Combining model checking and deduction for i/o-automata. In *Tools and Algorithms for Construction and Analysis of Systems*, pages 1–16, 1995.
20. S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In O. Grumberg, editor, *Proc. 9th INternational Conference on Computer Aided Verification (CAV'97)*, volume 1254, pages 72–83. Springer Verlag, 1997.
21. Natarajan Shankar. Combining theorem proving and model checking through symbolic analysis. In *CONCUR'00: Concurrency Theory*, number 1877 in Lecture Notes in Computer Science, pages 1–16, State College, PA, aug 2000. Springer-Verlag.
22. A. Pnueli Z. Manna. *The Temporal Logic of Reactive and Concurrent Systems.* Springer-Verlag, 1992.