Model checking for infinite state systems using data abstraction, assumption-commitment style reasoning and theorem proving

Jürgen Dingel[1] and Thomas Filkorn[2]

[1] Carnegie Mellon University, School of Computer Science, Pittsburgh, USA, jurgend@cs.cmu.edu
[2] Siemens AG, Corporate Research and Development, München, Germany, filkorn@zfe.siemens.de

Abstract. A method combining data abstraction, model checking and theorem proving is presented. It provides a semi-automatic, formal framework for proving arbitrary linear time temporal logic properties of infinite state reactive systems. The paper contains a complete case study to prove safety and liveness of an implementation of a scheduler for the readers/writers problem which uses unbounded queues and sets. We argue that the proposed framework could be automated to a very large extent making this approach feasible in an industrial environment.

1 Introduction

The applicability of model checking [CE81, QS81, CES86, BCM+92] is limited by the necessity to model the behaviour of the system as a finite state machine. Hard- and software systems very often contain possibly infinite data values (e.g., integer values or recursive data structures like lists) or data which are much too large to be represented efficiently by a finite set of values (e.g., messages in protocols containing more than 100 bytes). Consequently, the first step in the application of model checking techniques to practical examples is to model the original system as a finite state system, which typically involves some kind of abstraction. Recently, [CGL92, Lon93, GL93, Gra94] have proposed formal abstractions to be used to model check very large state spaces. Here, given abstraction mappings for each of the domains of the state space, an (approximation of the) corresponding abstract system is computed such that each path (behaviour) in the concrete system has at least one "corresponding" path in the abstract system. Properties are expressed as formulae over a logic in which the existence of a certain path cannot be expressed. For instance, the linear time logic LTL and the branching time logic ACTL only allow for formulae which state that a property holds for all paths of the system. Soundness then is ensured by the fact that whenever a formula holds in the abstract system, it also holds in the concrete system.

[1] The first author would like to thank Siemens AG for making this research possible with a summer internship.

One advantage of this approach is that it is fully automatic. Given the abstraction mappings no further user interaction is required. One disadvantage is that in case a property does not hold in the abstract system the user can only change the abstraction mapping and try again. Furthermore, although the property holds in the concrete system there might not be an abstraction such that the property also holds in the abstract system or the user may not find it. When moving to infinite state systems, this disadvantage is even more dramatic: Because an infinite state space has to be mapped onto a finite one, all abstractions are bound to add a relatively large amount of behaviour, which has no counterpart in the concrete system. Consequently, especially many interesting liveness properties, which do hold in the concrete system, will fail in the abstract system and will not be provable at all using the above method.

Our method trades automation for an increased flexibility in finding the right abstraction. More specifically, in contrast to previous approaches, the user may be able to prove some or maybe all properties he/she is interested in using a very simple and natural abstraction. However, the proof may require some user interaction and expertise. In our framework the linear time logic LTL was used as a specification logic, but the approach would work similarly with the branching time logic ACTL. Given an description of an infinite state reactive system, e.g., the description of a protocol, and data abstractions for each of the domains of the state space (e.g., integers, sets, queues) the abstract, finite state system is computed automatically. A data abstraction consists of a mapping from concrete to abstract values and abstract versions for the operations on the values. In the description of the original system we replace all datatype declarations and operations by their abstract counterparts to obtain a description of the abstract system, from which a transition system can be generated automatically. Then, having initialized the current set of assumptions with the empty set, we use a model checker which supports assumption-commitment style reasoning and generates counterexamples for the following iterative process: If the abstract system satisfies the commitment under the current assumptions we are done and we can proceed to discharge the assumptions in the concrete system. If the commitment is not met, we analyze the generated counterexample to determine whether it corresponds to a real program error in the concrete system or to an unrealistic sequence impossible in the original program. In the first case, we debug the concrete system and start again. In the last case, we find an assumption excluding this counterexample and others of the same kind, add it to the current set of assumptions and check if the system now satisfies the commitment under these augmented assumptions. During this process one might, of course, also decide that the data abstraction needs to be changed and start all over again. Each of the assumptions will restrict the behaviour of either the environment or the system itself. To ensure that the system assumptions only exclude behaviour of the abstract system added by the abstraction, they have to be discharged in the concrete system. The environment assumptions have to be discharged in the environment the concrete system is running in. Since the concrete system has infinitely many states, a theorem prover has to be used to show that it satisfies the system assumptions. Moreover, the theorem prover is also employed to establish a correspondence (homomorphism) between the concrete and the abstract system. Theorem provers typically require substantial user interaction and expertise, which is the main reason why they are not widely used in industrial environments. However, we expect the proof obligations for the theorem prover to be relatively easy, making our approach feasible in an industrial environment. This is for three reasons:

1. In reactive systems like industrial control tasks or protocols the control aspects are dominant and safety critical, whereas data processing is often less important. Thus, most specified properties will be mostly control-oriented and data aspects will only play a minor role. Given this assumption, the verification problem can be split into a finite state control problem, which is solved *automatically* by model checking, and the verification of properties about the data part for which theorem proving must be used. Since the specified properties will be mostly control-oriented we expect that most of the work (verification and uncovering of errors) will be done by model checking, whereas the remaining assumptions should be so simple, that they can be handled easily using a theorem prover.
2. Since a data abstraction is given as a surjective function mapping from concrete data values to abstract ones, the task of proving homomorphy between the concrete and the abstract system reduces to proving that this abstraction mapping is homomorphic with respect to both the concrete functions manipulating these data and their abstract counterparts. In other words, the obligation to prove the homomorphism can be localized to these functions, because they constitute the only difference between the concrete and abstract system.
3. While working with the proposed method, we found that some basic induction rules suffice to break the problem of proving the temporal logic assumptions in the concrete system down to simple Hoare-triples, i.e., to proving implications of the kind "if a precondition holds and one step of the reactive program is executed then a certain postcondition is guaranteed". These Hoare-triples can readily be verified using a first order theorem prover.

We used our method to prove safety and liveness of a scheduler for the readers/writers problem which uses unbounded sets and queues. The implementation was described in an VHDL-like imperative language, and for the verification we used the SVE [FSS$^+$94] model checker and the first order theorem prover SEDUCT [SBN94]. In our example the theorem proving part required substantial user interaction. However, we believe that mayor improvements could be made here resulting in a much higher degree of automation. Our results suggest that the proposed method can help to make formal verification more feasible in an industrial environment.

The paper is organized as follows. Section 2 introduces the necessary theoretical background. A description of the readers/writers system and the used data abstraction is given in Sections 3 and 4. Section 5 demonstrates the usage of our method for the verification of essential safety and liveness properties. The discharge of the assumptions is carried out in Section 6. [DF94] is the full version of this paper containing complete program and proof listings.

2 Background

Definition 1. A *transition system* M is a tuple $(\Sigma_M, \rightarrow_M, I_M, P_M, v_M)$ where Σ_M is the set of states, $\rightarrow_M$ the transition relation with $\rightarrow_M \subseteq \Sigma_M \times \Sigma_M$, $I_M \subseteq \Sigma_M$ the set of initial states, P_M the set of propositions and v_M a labeling function with $v_M : \Sigma_M \rightarrow 2^{P_M}$. □

A *path* π *in* M is an infinite sequence $s_0, s_1, \ldots$ such that $s_i \rightarrow_M s_{i+1}$ for all $i \geq 0$. If π is the path $s_0, s_1, \ldots$, then π_i denotes the state s_i and π^i denotes the suffix $s_i, s_{i+1}, \ldots$.

Definition 2. Given some set P of atomic propositions and assuming $p \in P$, the set of LTL formulas is inductively defined as:

$$\phi ::= p \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \mathsf{X}(\phi) \mid \phi_1 \, \mathsf{U} \, \phi_2$$

Other formulas can be introduced as abbreviations in the usual way: $\phi_1 \vee \phi_2$ abbreviates $\neg(\neg\phi_1 \wedge \neg\phi_2)$, $\phi_1 \rightarrow \phi_2$ abbreviates $\neg\phi_1 \vee \phi_2$, true abbreviates $p \vee \neg p$ and false abbreviates $\neg$true. The temporal operator $\mathsf{F}(\phi)$ abbreviates true $\mathsf{U}\ \phi$ and $\mathsf{G}(\phi)$ abbreviates $\neg\mathsf{F}(\neg\phi)$. The satisfaction relation $\models$ of a LTL formula with respect to a path π in the transition system M is inductively defined over the structure of the formula:

$(M, \pi) \models p$ if $p \in v_M(\pi_0)$
$(M, \pi) \models \neg\phi$ if not $(M, \pi) \models \phi$
$(M, \pi) \models \phi_1 \wedge \phi_2$ if $(M, \pi) \models \phi_1$ and $(M, \pi) \models \phi_2$
$(M, \pi) \models \mathsf{X}(\phi)$ if $(M, \pi^1) \models \phi$
$(M, \pi) \models \phi_1 \ \mathsf{U} \ \phi_2$ if $\exists k : \ (M, \pi^k) \models \phi_2$ and $(M, \pi^i) \models \phi_1$ for all $0 \leq i < k$

If $(M, \pi) \models \phi$ then we say that M *satisfies* ϕ *along* π. If for all paths π in M with $\pi_0 \in I_M$ we have $(M, \pi) \models \phi$ then we say that M *satisfies* ϕ, written $M \models \phi$. □

The soundness of verification methods involving abstractions typically hinges on the following property: Whenever the abstract system satisfies a formula, then the concrete system also satisfies this formula. Since LTL (ACTL) formulas are interpreted over *all* possible behaviours, it suffices to show that every behaviour of the concrete system has a corresponding behaviour in the abstract system. Because checking language containment is hard (especially if one transition system is infinite), we use the stronger notion of homomorphy instead.

Definition 3. Let M, M' be two transition systems. A surjective function $h : \Sigma_M \rightarrow \Sigma_{M'}$ is *homomorphic* with respect to M and M', if

1. $\forall s \in \Sigma_M : \ v_M(s) = v_{M'}(h(s))$ and
2. $\forall s, t \in \Sigma_M : \ s \rightarrow_M t$ implies $h(s) \rightarrow_{M'} h(t)$ and
3. $\forall s \in I_M : \ h(s) \in I_{M'}$ □

Note that homomorhpy implies a simulation relation, which is just "one half" of Milner's bisimulation [Mil89].

Theorem 4. *Let h be a homomorphic function with respect to two transition systems M and M' and ϕ an arbitrary LTL formula. Then*

$$M' \models \phi \text{ implies } M \models \phi$$

□

The theorem can be proved by structural induction over the formula and is based on the fact that homomorphy induces a simulation relation and thus ensures that for every path in M a "corresponding" path in M' can be found.

Using our method the user defines an abstraction mapping h and constructs an abstract system M'. Verification of a property C under environment assumptions A_E for the concrete system M is based on the following theorem, which is a direct consequence of Theorem 4.

Theorem 5. *Let M and M' be two transition systems and let the environment assumptions A_E, the assumptions on the abstract system A_A, and the commitment C be given as LTL formulas. If*

1. *$M' \models (A_A \wedge A_E) \rightarrow C$ and*
2. *$M \models A_A$ and*
3. *h is homomorphic with respect to M and M'*

then $M \models A_E \rightarrow C$. □

Informally, if the abstract system satisfies the commitment assuming A_A and A_E, and the assumptions A_A can be discharged in the concrete system, and the abstract system simulates the concrete, then the concrete system also satisfies the commitment assuming A_E. If the abstract system is finite state, the first condition can be verified using model checking. Since the concrete system in general is not finite state, the last two conditions have to be checked by means of a theorem prover.

3 A reader/writer system

We demonstrate our method with a scheduler, which is to synchronize read and write access to a shared resource. The example was chosen for various reasons: Firstly, it is small enough to enable us to easily evaluate the proposed method. Secondly, queues are a common datatype, which are often used in communication protocols for buffering communication between processes. Although in most practical cases the maximal number of the messages waiting in the queue is bounded, this bound is often too large to allow for the application of model checking directly, especially when the number of different messages is large. Thirdly, the scheduler problem is typical for the applications we have in mind, because the interesting properties are mostly control-oriented.

The scheduler SCHED receives read and write requests from other processes and grants the resource subject to the following conditions:

1. at most one process is writing.
2. reading and writing to the resource must be mutually exclusive.
3. every request will eventually be granted.

Conditions 1 and 2 are typical instances of safety properties whereas the last condition is a liveness property.

The syntax and semantics of our imperative implementation language is very close to VHDL. Note that we could have chosen any other imperative language. The programs were translated automatically into symbolic representations of transition systems [FPW92]. During our experiments we extended this translator to support additional features (e.g., in-line definition of properties) which are not standard in VHDL. All computation is assumed to be instantaneous except the **wait** statement. Execution of a **wait** makes changes in the input and output variables visible. So, a VHDL program can be thought of as a transition system in which a transition is given by the computation between two **wait** statements.

In the following the implementation is sketched. A complete listing can be found in the appendix. The scheduler has two input (cmdIn, procIn) and two output (cmdOut, procOut) variables. cmdIn is the incoming command. procIn carries the id of the process issuing the current command. After the input has been processed cmdOut and procOut are output. cmdOut describes the response of the system. If cmdOut is an acknowledgement, procOut is the id of the process receiving the acknowledgement. Internally, the system has one process p with 5 local variables. wr and ww range over queues and contain the ids of processes currently waiting for read or write access respectively. aw and ar are sets containing the ids of the processes currently reading or writing. The heart of the program is an infinite loop which first processes the inputs and then computes the output. More precisely, first, requests are enqueued and processes deactivated and then one of the pending requests is chosen for acknowledgement.

A feature of our implementation language worth mentioning is the ability to define the so-called property variables. These are program annotations for verification purposes and sometimes also called probes or history, virtual, or auxiliary variables. They can be thought of as boolean variables, which are set to false at the beginning of each cycle and are only assigned to at certain, user-determined points. For instance, the execution of the statement

property(awNotEmptyAfterRemove := (aw $\neq$ emptySet))

causes aw $\neq$ emptySet to be evaluated. The result is assigned to the property variable awNotEmptyAfterRemove. Property variables turned out to be necessary for formulating assumptions on the abstract system. The VHDL implementation corresponds to a transition system in the obvious way where a state is a the tuple of the values of all variables and properties: [wr, ww, ar, aw, p_1, ..., p_n] where the p_i denote the property variables used.

4 Abstractions

Given the implementation of the scheduler we would like to prove its correctness with respect to the afore mentioned safety and liveness conditions. We first have to define data abstractions for each of the domains in the state space (here: set, queue, pid). We start by considering abstraction mappings for queues.

We cannot just map the empty queue to some abstract value *emptyQueue* and a queue with at least one element to a different abstract value *nonEmptyQueue*, because we lose control over whose requests are being acknowledged. Thus, this abstraction would not allow us to express the liveness property correctly. The correct formulation of liveness requires us to focus on one particular process with, say, id x and to stipulate that a request of a x-process is always eventually acknowledged. This is expressed by the following formula ϕ:

$$\mathsf{G}(\mathsf{startRead}(\mathsf{x}) \rightarrow \mathsf{F}(\mathsf{ackRead}(\mathsf{x})))$$

This formula still allows for a x-process to starve other x-processes, but when we assume that process ids are unique this situation cannot arise. To verify the above specification, we could define a data abstraction on process ids, which maps all ids different from x to some abstract id *notx* and is the identity on x, and then check the formula in the resulting abstract system. Following this idea, we introduce a symbolic constant c as a generic process id and define an abstraction which maps all ids different from c to the abstract id *notc* and maps c to c. It remains to be shown that the corresponding abstract system SCHED'_{h_c} satisfies the formula $\phi[c/x]$. This technique is called symbolic abstraction in [CGL92]. The behaviour of the scheduler obviously does not depend on the process ids. The transition relation of the abstract system therefore is not parametric in c and all the abstract systems SCHED'_{h_x} are isomorphic. Thus, if the scheduler works correctly on x-processes, we can generalize and conclude that it handles all processes correctly. We note that, similar to symbolic abstractions, the applicability of our method does not rely on this special case.

To define the data abstractions, surjective mappings from concrete to abstract domains and the abstract counterparts of the operations have to be given. We start with the mappings.

Definition 6. The *abstraction mapping* h_c is defined by individual abstraction mappings for the components of the state space, which are also denoted by h_c. For the datatype **queue** the mapping is as follows: Let $Q' = \{emptyQueue, onlyc, fullnoc, fullandc\}$ be the set of the values for the abstracted queue and let Q denote the domain of the datatype **queue**. The abstraction mapping $h_c : Q \rightarrow Q'$ for the datatype queue is defined as:

$$h_c(q) = \begin{cases} emptyQueue & \text{if } q \text{ is the empty queue} \\ onlyc & \text{if } q \text{ is non-empty and contains only c-processes} \\ fullnoc & \text{if } q \text{ is non-empty and contains no c-processes} \\ fullandc & \text{if } q \text{ is non-empty and contains c- and notc-processes} \end{cases}$$

The abstraction mapping on **set** is defined analogously. The datatype **procInIdT** is dealt with as follows: Let *PID* be the set of process ids. The set $PID' = \{c, notc\}$ is the set of abstract values. The abstraction mapping for the process ids is given by:

$$h_c(pid) := \begin{cases} c & \text{if } pid = c \\ notc & \text{otherwise} \end{cases}$$

□

We want h_c as defined above to be an homomorphism with respect to the concrete operations *insertQueue* and *removeQueue* and the abstract operations *insertQueue'* and *removeQueue'*. One way to ensure this, is to define the abstract operations such that homomorphy of h_c is guaranteed. Nondeterminism introduced by the abstraction is modeled by nondeterministic operations, i.e., set-valued functions. More precisely, given an operation f, the abstraction h_c induces a homomorphic abstract operation f' given by: $f'(x') = \{y' \mid \exists x, y : h_c(x) = x' \text{ and } f(x) = y \text{ and } h_c(y) = y'\}$. Note that we assume behaviours to be infinite. Thus, all operations have to be total.

Definition 7. Let Q' be the abstracted values of the queue. The abstract operator *removeQueue'* $: Q' \to 2^{Q'}$ is defined as:

$$\begin{aligned}
removeQueue'(emptyQueue) &= \{emptyQueue, onlyc, fullnoc, fullandc\}\\
removeQueue'(onlyc) &= \{onlyc, emptyQueue\}\\
removeQueue'(fullnoc) &= \{fullnoc, emptyQueue\}\\
removeQueue'(fullandc) &= \{fullandc, fullnoc, onlyc\}
\end{aligned}$$

The abstract operator *insertQueue'* and the abstract operations on sets, *insertSet'* and *removeSet'*, are defined analogously. □

In the above definition, the application of an abstract operation (*removeQueue'*) to some bogus input (*emptyQueue*) may return any value. Erroneous input could also be dealt with more explicitly by introducing an error state. Unfortunately, such a formulation makes the theorem proving work much harder because, roughly speaking, domains with explicit error states are not inductively generated anymore.

The two definitions above constitute the data abstractions and define the abstract scheduler SCHED'_{h_c}.

Proposition 8. *h_c is a homomorphic mapping from* SCHED *to* SCHED'_{h_c}.

The data abstractions for the abstract datatypes **set** and **queue** define two "abstracted" abstract datatypes **set'** and **queue'**. We obtain the implementation of the abstract scheduler SCHED'_{h_c} by just importing **set'** and **queue'** instead of **set** and **queue**. In other words, due to modularity and data abstraction (in the program design sense) the change in the implementation can be localized to the abstract datatypes. Thus, given the abstracted version of the abstract datatypes we can easily generate a description of the abstract system and also automatically generate transition systems to be used for model checking.

5 Proof of Safety and Liveness

It is described how safety and liveness of the abstract system SCHED'_{h_c} were proven. Not surprisingly, liveness will turn out a lot harder to prove than safety. Throughout this section the finite state machine input to the model checker is the abstract scheduler SCHED'_{h_c} of Section 4.

5.1 Safety

The right formulation of the safety condition (condition 2 on page 6) is
Commitment: $\mathsf{G(aw = emptySet \lor ar = emptySet)}$
When running the model checker with no assumptions and the above commitment, a counterexample is output in which the environment issues an **`endWrite(c)`** without a previous write acknowledge. Such behaviours exhibiting meaningless input are excluded by the following environment assumption.
Assumption A_1:

$$\begin{aligned}\mathsf{G(}\ &\mathsf{(endRead(c) \rightarrow (ar = onlyc) \lor (ar = fullandc)) \land}\\ &\mathsf{(endWrite(c) \rightarrow (aw = onlyc) \lor (aw = fullandc)) \land}\\ &\mathsf{(endRead(notc) \rightarrow (ar= fullnoc) \lor (ar= fullandc)) \land}\\ &\mathsf{(endWrite(notc) \rightarrow (aw= fullnoc) \lor (aw= fullandc)))}\end{aligned}$$

This formula ensures that only currently active processes are able to release the resource. Under this assumption SCHED'_{h_c} satisfies the commitment.

5.2 Liveness for reading c-processes

Let the formula presented in Section 4 be the commitment for this subsection.
Commitment: $\mathsf{G(startRead(c) \rightarrow F(ackRead(c)))}$
Running the model checker with assumption A_1 and the above commitment we find that SCHED'_{h_c} can perform counterexample C_2 below. A counterexample will be represented by an infinite sequence of pairs where the two components denote the input and the corresponding output respectively, and the loop is indicated using the Kleene star.

Counterexample C_2:
```
startWrite(c),ackWrite(c)
startRead(c),idle(none)
( idle(c),idle(none) )*
```
Assumption A_2:
$\mathsf{G(ackWrite \rightarrow X(F(endWrite)))}$

Having sent a write acknowledgement, the system idles forever waiting for the writing process to release the resource by issuing an **`endWrite`** command. Thus, the resource is never released, preventing the read request from eventually being acknowledged. We have to make sure that whenever the resource is granted for writing that it is eventually released by the environment. This is a very typical situation, where the liveness of one system component depends on the liveness of its communication partners. Running the model checker with the conjunction of Assumption A_1 and A_2 yields the following counterexample.

Counterexample C_3:
```
startWrite(c),ackWrite(c)
startRead(c),idle(none)
( endWrite(c),idle(none) )*
```
Assumption A_3:
$\mathsf{G(\neg awNotEmptyAfterRemove)}$

In this sequence the command **`endWrite`** does not empty the set **`aw`** due to the non-determinism in the *removeSet'* operation modeling the fact that **`aw`** may have more than one element. However, the scheduler should only allow at most one active writer in the first place. Thus, assuming that the scheduler really does ensure this and that we only perform remove operations on non-empty sets it is safe to assume that the set **`aw`** is always empty after a remove.

We use the property variable **awNotEmptyAfterRemove** to express this. This property variable is true if and only if after a remove operation on the set **aw** the resulting set is not empty. A_3 is an assumption about the system itself and will be discharged later with the theorem prover. Assuming that SCHED'_{h_c} satisfies $\bigwedge_{i=1}^{3} A_i$ the following trace disproves the commitment.

Counterexample C_4:

```
startWrite(c),ackWrite(c)
startRead(c),idle(none)
( startRead(notc),idle(none)
endWrite(c),ackRead(notc)
startWrite(c),idle(none)
endRead(notc),ackWrite(c) )*
```

Assumption A_4:

$$\mathsf{G}(\mathsf{inserted(c,wr)} \wedge \mathsf{G}(\mathsf{F}(\mathsf{removed(wr)})) \rightarrow \mathsf{F}(\mathsf{removed(c,wr)}))$$

Here, the problem is that when **wr = onlyc** we can keep on inserting and removing a notc-process forever. In other words, the c-process waiting in **wr** is always "overtaken" by a notc-process. To remedy this we have to resort to a simple fact which for a concrete queue could easily be proved by induction over the number of its elements: Whenever there is a c-process waiting in queue **wr** and we remove processes from that queue infinitely often, then the c-process will eventually also be removed. The atomic propositions inserted(c,wr) and so on are implemented as property variables indicating that an insert is performed. Under the assumption that SCHED'_{h_c} satisfies $\bigwedge_{i=1}^{4} A_i$ the model-checker outputs the following counterexample.

Counterexample C_5:

```
startWrite(c),ackWrite(c)
startRead(c),idle(none)
startRead(notc),idle(none)
endWrite(c),ackRead(notc)
startWrite(c),idle(none)
( idle(c),idle(none) )*
```

Assumption A_5:

$$\mathsf{G}(\mathsf{ar} \neq \mathsf{emptySet} \rightarrow \mathsf{F}(\mathsf{endRead}))$$

In this case it is again the environment which "misbehaves": A notc-process is granted the resource for reading, but never releases it, thus starving the waiting c-process. This can be ruled out by requiring that whenever there are active readers, the environment will eventually issue an **endRead**. However, even under the assumption $\bigwedge_{i=1}^{5} A_i$ SCHED'_{h_c} still does not satisfy lifeness:

Counterexample C_6:

```
startWrite(c),ackWrite(c)
startRead(c),idle(none)
startRead(notc),idle(none)
endWrite(c),ackRead(notc)
startWrite(c),idle(none)
( endRead(notc),idle(none) )*
```

Assumption A_6:

$$\mathsf{G}(\mathsf{G}(\neg\mathsf{inserted(ar)}) \wedge \mathsf{G}(\mathsf{F}(\mathsf{removed(ar)})) \rightarrow \mathsf{F}(\mathsf{ar}{=}\ \mathsf{emptySet}))$$

Counterexample C_6 differs from C_3 only in that a c-process is now being starved in the queue for waiting readers. Again, the statement **endRead(notc)** in the loop does not remove all notc-processes the set **ar** due to the nondeterminism in *removeQueue'*. In contrast to C_3 more than one process can be reading. However, we can resort to another simple fact about sets: If we perform an infinite number of remove operations and no insert, then the set will eventually become empty. Now, under the assumption $\bigwedge_{i=1}^{6} A_i$ SCHED'_{h_c} finally does satisfy liveness and the model checker terminates with "yes".

5.3 Proof of liveness for writing c-processes

We also have to prove that a write request of a c-process is eventually acknowledged.

Commitment: $\mathsf{G}(\mathsf{startWrite(c)} \rightarrow \mathsf{F}(\mathsf{ackWrite(c)}))$

This commitment fails under assumption $\bigwedge_{i=1}^{6} A_i$ because, analogous to counterexample C_4, a c-process can always be overtaken by a notc-process in **ww**. It suffices to stipulate that whenever there is at least one c-process waiting for writing and we keep activating then the waiting c-process will eventually also be activated.

Assumption A_7: $\mathsf{G}(\mathsf{inserted(c,ww)} \wedge \mathsf{G}(\mathsf{F}(\mathsf{removed(ww)})) \rightarrow \mathsf{F}(\mathsf{removed(c,ww)}))$

Assuming that SCHED'_{h_c} satisfies $\bigwedge_{i=1}^{7} A_i$ it also satisfies lifeness for writing processes.

Resume

Using the above method, a simple and natural data abstraction was sufficient to prove safety and liveness. Note that using the standard approach as proposed in [CGL92, Lon93, GL93, Gra94] the above abstraction would not have allowed for a successful proof. In an iterative process the counterexamples generated by the model checker directly lead to the assumptions necessary for completion of the proof. Without the debugging capabilities of the model checker it would have been very difficult for a user to come up with the necessary assumptions and most likely they would have been too strong. During this verification process we also got a much deeper understanding of the program and the implicit assumptions made about the environment which in itself can be a valuable result in industrial applications.

6 Discharging the assumptions

Obviously, it is only the system assumptions (A_3,A_4,A_6,A_7) we can discharge in the concrete system. The environment assumptions (A_1,A_2,A_5) would have to be discharged with an environment using the scheduler. To discharge A_1, for example, we could show that the environment only releases the resource if it has previously acquired it. Similarly for assumption A_2 and A_5.

We start by proving that SCHED satisfies assumption A_4. We will be able to reuse the proof for assumptions A_6 and A_7 to a very large extend.

Assumption A_4: $\mathsf{G}(\mathsf{inserted(c,wr)} \wedge \mathsf{G}(\mathsf{F}(\mathsf{removed(wr)})) \rightarrow \mathsf{F}(\mathsf{removed(c,wr)}))$

Proof. Let $\pi = s_0, s_1, \ldots$ be a path in SCHED. Assume that in state s_i we have:

$$\mathsf{inserted(c,wr)} \land \mathsf{G(F(removed(wr)))} \tag{1}$$

We need to show that π^i also satisfies F(removed(c,wr)). The idea is to introduce a measure m_c^Q denoting the number of processes in front of a process c currently in the queue Q. From (1) we will conclude that m_c^Q will be never incremented, but infinitely often decremented, and so it must become zero after a finite number of steps. To make this formal, we need some lemmas.

Lemma 9. *The following Hoare-triple is valid:*

$$\{m_c^Q = 0\} \quad \texttt{removeQueue(Q)} \quad \{\mathsf{removed(c,Q)}\}$$

This states that if no process is in front of a c-process and a `removeQueue` operation is performed, the c-process is removed. The next lemma states that a `removeQueue` operation decreases the number of processes in front of a c-process.

Lemma 10. *For all m, $n \geq 1$ the following Hoare-triple is valid:*

$$\{m_c^Q = n\} \quad \texttt{removeQueue(Q)} \quad \{m_c^Q < n\}$$

We also need a lemma saying that m_c^Q is preserved by all transitions not involving `removeQueue(Q)`.

Lemma 11. *For all operations* S *except* `removeQueue(Q)`*:*

$$\{m_c^Q = n\} \quad \mathsf{S} \quad \{m_c^Q = n\}$$

All of these lemmas were proven in a straightforward fashion using the first order theorem prover SEDUCT [SBN94]. Using Lemma 10 and 11 we get the following corollary.

Corollary 12. *m_c^Q canot be increased, i.e., for all $n \geq 1$, there is no transition* S *such that*

$$\{m_c^Q = n\} \quad \mathsf{S} \quad \{m_c^Q > n\}.$$

To complete the proof we introduce the following induction schema which is provable by induction over m:

$$\frac{m \in \mathbf{N} \land \mathsf{G(notup}(m)) \land \mathsf{G(F(down}(m)))}{\mathsf{F}(m = 0)} \tag{2}$$

where

$$\mathsf{notup}(m) \overset{def}{=} (m \geq 1 \land m = n) \rightarrow \mathsf{X}(m \leq n)$$

$$\mathsf{down}(m) \overset{def}{=} (m \geq 1 \land m = n) \rightarrow \mathsf{X}(m < n)$$

We apply the rule with the following instantiations:

1. $m \overset{def}{=} m_c^{wr}$
2. inserted(c,wr) $\mapsto m_c^{wr} \in \mathbf{N}$ (immediate)
3. removed(wr) $\mapsto$ down(m_c^{wr}) (Lemma 10)
4. G(notup(m_c^{wr})) (Corollary 12)

The Hoare-triples proved above for single statements are combined to derive the corresponding Hoare-triples for the whole part of the program between the two wait statements. Using assumption (1), we can conclude that π^i satisfies $\mathsf{F}(m_c^{wr} = 0)$. In other words, there is a state s_j on π^i such that $m_c^{wr} = 0$ in s_j. Let s_k be the first state after s_j satisfying removed(wr) (which must exist due to (1)). Using Lemma 11, s_k must also satisfy $m_c^{wr} = 0$. Thus, s_k satisfies removed(wr) $\wedge$ $m_c^{wr} = 0$. But then, using Lemma 9, removed(c,wr) also holds. ■

For assumption A_7 we can employ the same reasoning except that the queue wr in the above argument is replaced by the queue ww. For assumption A_6 we have to modify the above argument slightly for sets, where we use the cardinality as a measure and lemmas analogous to Lemma 10 and 11.
Assumption A_6: $\mathsf{G}(\mathsf{G}(\neg\mathsf{inserted(ar)}) \wedge \mathsf{G}(\mathsf{F}(\mathsf{removed(ar)})) \rightarrow \mathsf{F}(\mathsf{ar} = \mathsf{emptyQueue}))$

Proof. We apply rule (2) with the following instantiations:

1. $m \overset{def}{=} \mid \mathsf{ar} \mid$, i.e., the number of elements of ar.
2. $\mid \mathsf{ar} \mid \in \mathbb{N}$ (by definition)
3. removed(ar) $\mapsto$ down($\mid$ ar $\mid$) (equivalent of Lemma 10)
4. $\neg$inserted(ar) $\mapsto$ notup($\mid$ ar $\mid$) (equivalent of Lemma 11)

and conclude that $\mathsf{F}(m = 0)$ and thus also F(ar = emptyQueue) hold along the current path. ■

Assumption A_3 unfortunately cannot be proved in the above fashion.
Assumption A_3: G($\neg$awNotEmptyAfterRemove)
This assumption is a statement about the entire program and how it affects the set of active writers, **aw**, and not just about the implementation of the abstract datatypes. This assumption follows directly from a prove of safety condition 1 (at most one process is writing, i.e. $|aw| \leq 1$). This invariant was proved by proving the Hoare-triple $\{|aw| \leq 1\}$ P $\{|aw| \leq 1\}$. This invariant could have also be proved using model checking with an abstraction which maps the elements of **aw** to three values, *empty, oneElement, manyElements.*

7 Conclusion and Further Work

We have proposed a method which combines data abstraction with the use of model checking and theorem proving to provide a formal framework for proving arbitrary linear time temporal logic properties of infinite state reactive systems. One of its essential features is that the mere possibility of proving a property about an infinite state system does not depend on the "granularity of the abstraction", because the behaviour added by the abstraction can successively be ruled out. This gives the user more freedom when choosing the abstraction: In contrast to [CGL92] almost any abstraction can be used and the same abstraction can be employed for different properties. However, with a very coarse abstraction the necessary assumptions might be so strong that discharging them is just as hard as proving the original property. Moreover, the method is also applicable when the property fails in the abstract system. In contrast to [CGL92] we do not have to look for a different abstraction which does not interfere with the property.

For a very restricted class of so called "data-independent" programs Wolper [Wol86] showed how to map these programs and special classes of LTL specifications to finite data domains. In his approach a specification is valid in the finite data domain if and only if it is valid in the original data domain. Our method does not only handle these kinds of data abstractions but also a much broader class of programs and abstractions, because of the possibility to introduce assumptions.

It would obviously be of great value to extend automatic verification to infinite state systems as much as possible. Some kind of combination of model checking and theorem proving seems very promising in this respect [Hun93]. However, it is not clear how the overall task should be split among these two components and how they should interact. For all of these approaches the degree of automation possible is a useful criterion to determine its practical applicability and to predict its success.

We have sketched the architecture of a verification system, which, we believe, could be automated to a large extend without sacrificing too much generality. More specifically, it should be possible to automate the following phases: computation of the abstract versions of the abstract data types involved in the abstraction; separation of the environment and the system assumptions; discharge of simple assumptions. However, the synthesis of temporal logic formulae from counterexamples and discharge of complex system assumptions might be impossible to automate without heavy restrictions.

We have demonstrated the feasibility of our ideas with the complete case study of a simple yet common problem.

Further work

The theorem prover and its interface with the model checker leave a lot of room for improvement. Firstly, there seems to be an unnecessarily large gap between the implementation (here, the VHDL-like program and its temporal logic specifications) on the one hand and its logic representation (here, first order logic with equality) on the other hand. Logical frameworks like LF or ELF [Pfe94] may allow for the development and implementation of a logic in which programs and temporal properties could be represented more uniformly. This logic could have rules like the one presented in Section 6 built into its deductive engine. Secondly, the task of discharging the system assumptions would be greatly facilitated if for each abstract data type the theorem prover had access to a library of lemmas, induction rules etc. describing that data type. Moreover, it would be illuminating to apply the method to a larger practical example (e.g., a communication protocol with data).

References

[BCM+92] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, 1992.

[CE81] E.M. Clarke and E.A. Emerson. Synthesis of synchronization skeletons for branching time temporal logic. In *Logic of programs: Workshop, Yorktown Heights, NY, May 1981*, volume LNCS 131. Springer Verlag, 1981.

[CES86] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Programming Languages and Systems*, 1(2):244–263, 1986.

[CGL92] E.M. Clarke, O. Grumberg, and D.E. Long. Model checking and abstraction. In *Proceedings of the 19th ACM Symposium on Principles of Programming Languages*, pages 343–354, New York, 1992. ACM Press.

[DF94] J. Dingel and T. Filkorn. Model checking for infinite state systems using data abstraction, assumption-commitment style reasoning and theorem proving. Technical Report ZFE BT SE 1-?, Siemens AG, Corporate Research and Development, Munich, 1994. Draft.

[FPW92] T. Filkorn, M. Payer, and P. Warkentin. Symbolic verification of sequential circuits synthesized with CALLAS. In D. Gajski, editor, *Proc. 6th International Workshop on High-Level Synthesis*, pages 344–353, Laguna Nigel, CA, U.S.A., November 1992. ACM/IEEE.

[FSS+94] Th. Filkorn, H.A. Schneider, A. Scholz, A. Strasser, and P. Warkentin. SVE User's Guide. Technical Report ZFE BT SE 1-SVE-1, Siemens AG, Corporate Research and Development, Munich, 1994.

[GL93] S. Graf and C. Loiseaux. A tool for symbolic program verification and abstraction. In *Computer Aided Verification, 5th International Conference*, volume LNCS 697, pages 71–84. Springer Verlag, 1993.

[Gra94] S. Graf. Verification of a distributed cache memory by using abstractions. In *Computer Aided Verification, 6th International Conference*, volume LNCS 818, pages 207–219. Springer Verlag, 1994.

[Hun93] Hardi Hungar. Combining model checking and theorem proving to verify parallel processes. In *Computer Aided Verification, 5th International Conference*, volume LNCS 697, pages 154–165. Springer Verlag, 1993.

[Lon93] David Long. *Model Checking, Abstraction, and Compositional Verification.* PhD thesis, Carnegie Mellon University, July 1993.

[Mil89] Robin Milner. *Communication and Concurrency.* Prentise Hall, 1989.

[Pfe94] Frank Pfenning. Elf: A meta-language for deductive systems. In *Proceedings of CADE-12*, volume LNAI 814, pages 811–815. Springer Verlag, 1994.

[QS81] J. Quielle and J. Sifakis. Synthesis of synchronization skeletons for branching time temporal logic. In *Proceedings of the 5th International Symposium in Programming*, volume LNCS 137. Springer Verlag, 1981.

[SBN94] Karl Stroetmann and Claus Bendix Nielsen, editors. *A Guide to* SEDUCT. Siemens AG, Munich, Germany, 1994.

[Wol86] Pierre Wolper. Expressing interesting properties of programs in propositional temporal logic. In *Proceedings of Principles of Programming Languages 1986*, pages 184–193, 1986.

A Implementation of SCHED

This section lists the implementation of our concrete scheduler. We assume that the file SchedTypes.vhdl implements the abstract datatypes set and queue and thus correctly defines the functions removeSet, insertSet, removeQueue, insertQueue, the predicates isEmptyS and isEmptyQ and the constants emptySet and emptyQueue. Moreover, we assume that the types cmdInT, cmdOutT , procInIdT, procOutIdT, and preferT are defined as indicated. prefer is a variable indicating whether a reader or a writer has just released the resource and thus allows for readers and writers being acknowledged alternately.

```
#include "SchedTypes.vhdl"
ENTITY rw IS PORT
  clk : IN bit;
  cmdIn : IN cmdInT;                /* cmdInT = {startRead,startWrite,endRead,endWrite,idle} */
  cmdOut : OUT cmdOutT;                                /* cmdOutT = {ackRead,ackWrite,idle} */
  procIn : IN procInIdT;                                                   /* process ids */
  procOut : OUT procOutIdT);                               /* process ids including "none" */
END rw;
ARCHITECTURE be OF rw IS
BEGIN p: PROCESS
  VARIABLE aw,ar : set;
  VARIABLE ww,wr : queue;
  VARIABLE prefer : preferT;                          /* preferT = {none,readers,writers} */
  BEGIN
    ar := emptySet; aw := emptySet; wr := emptyQueue; ww := emptyQueue;
    main_loop: LOOP
      wait;                                                                 /* clock tick */
      prefer := none; procOut := none;                                  /* default output */
      IF (cmdIn = startRead) THEN insertQueue(wr,procIn);
        ELSIF (cmdIn = endRead) THEN
          prefer := writers; removeSet(ar,procIn);                 /* assume( procIn in ar ) */
          ELSIF (cmdIn = startWrite) THEN insertQueue(ww,procIn);
          ELSIF (cmdIn = endWrite) THEN
            prefer := readers; removeSet(aw,procOut);              /* assume( procIn in aw ) */
            property(awNotEmptyAfterRemove := (aw \ = emptySet));
          END IF;
          cmdOut := idle;                                               /* default output */
          IF (prefer = readers) THEN                                  /* activate readers */
            IF ( isEmptyS(aw) ∧ ~isEmptyQ(wr) ) THEN
              removeQueue(wr,procOut); insertSet(ar,procOut); cmdOut := ackRead;
            END IF;
          END IF;
          /* activate writers, but only if no output has been determined yet */
          IF (cmdOut = idle ∧ isEmptyS(ar) ∧ isEmptyS(aw) ∧ ~isEmptyQ(ww)) THEN
            removeQueue(ww,procOut); insertSet(aw,procOut); cmdOut := ackWrite;
          END IF;
          /* activate readers, but only if no output has been determined yet */
          IF (cmdOut = idle ∧ isEmptyS(aw) ∧ isEmptyQ(ww) ∧ ~isEmptyQ(wr)) THEN
            removeQueue(wr,procOut); insertSet(ar,procOut); cmdOut := ackRead;
          END IF;
      END LOOP main_loop;
  END PROCESS;
END be;
```