Combining Theorem Proving and Model Checking through Symbolic Analysis *

Invited Paper at CONCUR 2000

Natarajan Shankar

Computer Science Laboratory
SRI International
Menlo Park CA 94025 USA
{shankar}@csl.sri.com
URL: http://www.csl.sri.com/~shankar/
Phone: +1 (650) 859-5272

Abstract. Automated verification of concurrent systems is hindered by the fact that the state spaces are either infinite or too large for model checking, and the case analysis usually defeats theorem proving. Combinations of the two techniques have been tried with varying degrees of success. We argue for a specific combination where theorem proving is used to reduce verification problems to finite-state form, and model checking is used to explore properties of these reductions. This decomposition of the verification task forms the basis of the Symbolic Analysis Laboratory (SAL), a framework for combining different analysis tools for transition systems via a common intermediate language. We demonstrate how symbolic analysis can be an effective methodology for combining deduction and exploration.[1]

The verification of large-scale concurrent systems poses a difficult challenge in spite of the substantial recent progress in computer-aided verification. Technologies based on model checking [CGP99] can typically handle systems with states that are no larger than about a hundred bits. Techniques such as symmetry and partial-order reductions, partitioned transition relations, infinite-state model checking, represent important advances toward ameliorating state explosion, but they have not dramatically increased the overall effectiveness of automated verification. Model checking does have one advantage: it needs only a modest amount of human guidance in terms of the problem description, possible variable orderings, and manually guided abstractions. Verification based on theorem proving, on the other hand, requires careful human control by way of suitable intermediate assertions, invariants, lemmas, and proofs. Can automated verification ever combine the automation of model checking with the generality of theorem proving?

* This work was funded by DARPA Contract No. F30602-96-C-0204 Order No. D855 and NSF Grants No. CCR-9712383 and CCR-9509931.

[1] The SAL project is a collaborative effort between Stanford University, SRI International, and the University of California, Berkeley.

It has often been argued that model checking and theorem proving could be combined so that the former is applied to control-intensive properties while the latter is invoked on data-intensive properties. Achieving an integration of theorem proving and model checking is not hard. Both techniques verify claims that look similar and it is possible to view model checking as a decision procedure for a well-defined fragment of a specification logic [RSS95]. However, most systems contain a rich interaction between control and data so that there is no simple decomposition between data-intensive and control-intensive properties.

For the purpose of this paper, we view model checking as a technique for the verification of temporal properties of a program based on the exhaustive exploration of a transition graph represented in explicit or symbolic form. Model checking methods typically use graph algorithms, automata-theoretic constructions, or finite fixed point computations. Theorem proving is usually based on formalisms such as first-order or higher-order logic, and employs proof techniques such as induction, rewriting, simplification, and the use of decision procedures. Some infinite-state verifiers and semi-decision procedures can be classified as both deductive and model checking techniques, but this ambiguity can be overlooked for the present discussion.

We make several points regarding the use of theorem proving and model checking in the automated verification of concurrent systems:

1. *Correctness is over-rated.* The objective of verification is analysis, i.e., the accretion of useful observations regarding a system. Verifying correctness is an important form of analysis, but correctness is usually a big property of a system that is demonstrated by building on lots of small observations. If these small observations could be cheaply obtained, then the demonstration of larger properties would also be greatly simplified. The main drawback of correctness is its exactitude. The verification of a correctness claim can only either fail or succeed. There is no room for approximate answers or partial information.
2. *Theorem proving is under-rated.* Deduction remains the most appropriate technology for obtaining insightful, general, and reusable automation in the analysis of systems, particularly those that are too complex to be analyzed by a blunt instrument like model checking. Theorem proving can exploit the mathematical properties of the control and data structures underlying an algorithm in their fullest generality and abstractness
3. *Theorem proving and model checking are very similar techniques.* In the verification of transition systems, both techniques employ some representation for program assertions, they compute the image of the transition relation with respect to these assertions, and usually try to compute the least, greatest, or some intermediate fixed point assertion for the transition relation. The difference is that in theorem proving,
   - The image constructions are usually more complicated since they involve quantification in domains where quantifier elimination is either costly or impossible.
   - The least and greatest fixed points can seldom be effectively computed and human guidance is needed to suggest an intermediate fixed point.

  - Showing that one assertion is the consequence of another is typically undecidable and requires the use of lemmas and human insight.
4. *Theorem proving and model checking can be usefully integrated.* Such an integration requires a methodology that decomposes the verification task so that
   - *Deduction is used to construct valid finite-state abstractions of a system.* The construction of a property-preserving abstraction generates simple proof obligations that can be discharged, often fully automatically, using a theorem prover. These are typically assertions of the form: if property p holds in a state s from which there is a transition R to a state s', then property q holds in s'. Similar proof obligations arise during verification (in the form of verification conditions) but these are usually not valid and the assertions have to be strengthened in order to obtain provable verification conditions. While theorem proving is useful for examining the local consequence of properties, it is not very effective at deducing global consequences over a large program or around an iterative loop. Such computations can be extremely inefficient and the computation of fixed points around a loop rarely terminates.
   - *Exploration by means of model checking is used to calculate global properties of such abstractions.* This means that model checking is not used merely to validate or refute putative properties but is actually used to calculate interesting invariants that can be extracted from the reachability predicate or its approximations. Finite-state exploration of large structures can also be inefficient but it is much easier to make finite-state computations converge efficiently.
   - *Deduction is used to propagate the consequences of such properties.* For example, model checking on a finite-state abstraction might reveal an assertion $x > 5$ to hold at a program point simply because it was true initially and none of the intermediate transitions affected the value of x. If the program point has a successor state that can only be reached by a transition that increments x by 2, then we know that this successor state must satisfy the assertion $x > 7$. Such a consequence is easily deduced by theorem proving.

In summary, we advocate a verification methodology where deduction is employed in the local reasoning steps such as validating abstractions and propagating known properties, whereas model checking is used for deriving global consequences. In contrast, early attempts to integrate theorem proving and model checking were directed at using model checking as a decision procedure within a theorem prover. These attempts were not all that successful because it is not common to find finite-state subgoals within an infinite-state deductive verification.

1 Background

We review some of the background and previous work in the combined use of theorem proving and model checking techniques.

1.1 Model Checking as a Decision Procedure

Joyce and Seger combined the theorem prover HOL [GM93] with the symbolic trajectory evaluation tool Voss [JS93] by treating the circuits verified by Voss as uninterpreted constants in HOL. This integration is somewhat *ad hoc* since the definitions of the circuits verified by Voss are not available to HOL. Dingel and Filkorn [DF95] use a model checker to establish assume–guarantee properties of components and a theorem prover to discharge the proof obligations that arise when two components are composed. Rajan, Shankar, and Srivas [RSS95] integrate a mu-calculus [Par76,BCM$^+$92] model checker [Jan93] as a decision procedure for a fragment of the PVS higher-order logic corresponding to a finite mu-calculus. While this integration smoothly incorporates CTL and LTL model checking into PVS, the work needed to reduce a problem into model-checkable form can be substantial. This integration has recently been extended with an algorithm for constructing finite-state abstractions of mu-calculus expressions [SS99].[2]

1.2 Extending Model Checking with Lightweight Theorem Proving

Several alternative approaches to the integration of model checking and theorem proving have emerged in recent years. Some of these have taken the approach of supplementing a model checker with a proof assistant that provides rules for decomposing a verification goal into model-checkable subgoals. McMillan [McM99] in his work with Cadence SMV has extended the SMV model checker with the following decomposition rules that are used to reduce infinite-state systems to model-checkable finite-state ones.

1. *Temporal splitting*: Transforms a goal of the form $\Box(\forall i : A)$ into $\Box v = i \supset A$ for each i.
2. *Symmetry reduction*: Typically, the system being verified and the property are symmetric in the choice of i so that proving $\Box v = i \supset A$ for a single specific value for i is equivalent to proving it for each i. Examples of such symmetric choices include the memory address or the processor in the verification of multiprocessor cache consistency.
3. *Data abstraction*: Large or infinite datatypes can be reduced to small finite datatypes by suitably reinterpreting the operations on these datatypes. For example, with respect to the choice of i in temporal splitting, the remaining values of the datatype can be abstracted by a single value *non-i*.
4. *Compositional verification*: The verification of $P \| Q \models A \wedge B$ is decomposed as $P \models \neg(B\ \mathbf{U}\ \neg A)$ (B fails before A does) and $Q \models \neg(A\ \mathbf{U}\ \neg B)$. This allows different components to be separately verified up to time $t + 1$ by assuming the other components to be correct up to time t.

These and other proof techniques have been used to verify an out-of-order processor, a large cache coherence algorithm, and safety and liveness for a version of Lamport's N-process bakery algorithm for mutual exclusion [MQS00].

[2] These features are part of PVS 2.3 which is accessible at the URL `pvs.csl.sri.com`.

McMillan's approach is substantially deductive. The rules of inference, such as symmetry reduction and compositional verification, are specialized but quite powerful.

Seger [Seg98] has extended the Voss tool for symbolic trajectory evaluation with lightweight theorem proving. Symbolic trajectory evaluation (STE) which is a limited form of linear temporal logic model checking. A few simple proof rules are used to decompose proof obligations on the basis of the logical connectives such as conjunction, disjunction, and implication. These rules can be used to decompose a large model checking problem into smaller ones.

1.3 Abstraction and Model Checking

Abstraction has been studied in the context of model checking as a technique for reducing infinite-state or large finite-state models to finite-state models of manageable size [BBLS92,Kur94,CGL94,LGS+95,Dam96,BLO98].

Some of the work on abstraction is based on *data abstraction* where a variable X over a concrete datatype T is mapped to a variable x over an abstract type t. For example, a variable over the natural numbers could be replaced by a boolean variable representing the parity of its value. Clarke, Grumberg, and Long [CGL94] gave a simple criterion for abstractions that preserve $\forall CTL^{*+}$ properties. Let the concrete transition system be given by $\langle I_C, N_C\rangle$ where I_C is the initialization predicate and N_C is the next-state relation. Then the verification of a concrete judgement $\langle I_C, N_C\rangle \models P_C$ can be reduced by means of the abstraction function α to the verification of an abstract judgement $\langle I_A, N_A\rangle \models P_A$ provided

1. $I_C \sqsubseteq I_A \circ \alpha$
2. $N_C \sqsubseteq N_A \circ \langle \alpha, \alpha\rangle$
3. $P_A \circ \alpha \sqsubseteq P_C$

Data abstraction has the advantage that the abstract description can be statically constructed from the concrete program. The drawback is that many useful abstractions are on relations between variables rather than on individual variables.

Graf and Saïdi [SG97] introduced *predicate abstraction* as a way of replacing predicates or relations over a set of variables by the corresponding boolean variables. For example, given two variables x and y over the integers, and the predicate $x < y$ over these variables, predicate abstraction would replace the variables x and y by a boolean variable b that represents the behavior of the predicate.

The application of predicate abstraction makes significant use of theorem proving. Graf and Saïdi used predicate abstraction to construct an abstract reachability graph for a concrete program by a process of elimination. If a represent an abstract state, a' a putative successor, $\gamma(a)$ the concrete state corresponding to a, and $\gamma(a')$ the concrete state corresponding to a', then if

$$\gamma(a) \supset wp(P)(\neg(\gamma(a')))$$

is provable, the corresponding transition between a and a' can be ruled out.[3] However, if a proof attempt fails, the corresponding successor node can be conservatively included in the abstract reachability graph. Using predicate abstractions with the PVS theorem prover [ORS92], Graf and Saïdi [SG97] were able to verify a variant of the alternating bit protocol called the bounded retransmission protocol [HSV94]. Das, Dill, and Park [DDP99] extended this technique using the SVC decision procedures [BDL96] and were able to verify such impressive examples as the FLASH cache coherence protocol, and a cooperative garbage collector.

Predicate abstraction can also be used to construct an abstract transition relation instead of the abstract reachability graph. It is typically less expensive to construct the abstract transition relation since fewer proof obligations are generated, but it typically results in a coarser abstraction than one that is obtained by directly computing the abstract reachability graph. In the latter construction, information about the current set of abstract reachable states can be used to rule out unreachable successor states. Bensalem, Lakhnech, and Owre [BLO98] describe an abstraction tool called InVeSt that uses the elimination method to construct an abstract transition system from a concrete one in a compositional manner. Colon and Uribe [CU98] give another compositional method for constructing abstractions with the framework of the STeP theorem prover [MtSG95].

All of the above abstraction techniques preserve only $\forall CTL^{*+}$ properties, namely those in the positive fragment of CTL^* with universal path quantification. For more general calculi, criteria for abstractions that preserve CTL* [DGG94] and mu-calculus [LGS$^+$95], but these results are quite technical. Saïdi and Shankar [SS99] gave a simple method for constructing predicate abstractions over the full relational mu-calculus [Par76]. The two key observations in this work are:

1. The operators of the mu-calculus are monotonic with respect to upper and lower approximations.
2. The over-approximation of a literal (an atomic formula or its negation) can be efficiently computed in conjunctive normal form by using a theorem prover as an oracle.

Verification diagrams [MBSU99] can also be seen as a form of predicate abstraction. These diagrams employ graphs whose nodes are labeled by assertions and the edges correspond to program transitions within the diagram. Properties can be directly checked with respect to the verification diagram.

The primary advantage of predicate abstraction is that it is sufficient to guess a relevant predicates without having to guess the exact invariant in these predicates. For n predicates, the construction of the abstract transition system

[3] All programs are assumed to be total as transition system, i.e., the domain of the next-state relation is the set of all states. Thus, $wp(P)(A)$ is the set of states that have no transitions in P to states in $\neg A$. The dual notion $sp(P)(A)$ is the set of states reachable from some state in A by a transition of P.

generates of the order of 2^n proof obligations. The resulting abstract model can also be model checked in time that is exponential in n to yield useful invariants. With deduction, there are 2^{2^n} boolean functions that are candidate invariants in these n predicates so that it is harder to guess suitable invariants.

1.4 Automatic Invariant Generation

Automatic invariant generation has been studied since the 1970s [CH78,GW75,KM76,SI77]. This study has recently been revived through the work of Bjørner, Browne, and Manna [BBM97], and Bensalem, Lakhnech, and Saïdi [BLS96,Saï96,BL99].

The strongest invariant of a transition system P is given by the least fixed point starting from the initial states of P of the strongest postcondition operator for P, $\mu X.I_P \vee sp(P)(X)$. If this computation terminates, it would yield the set of reachable states of P which is its strongest invariant. Unfortunately, the least fixed point computation rarely terminates for infinite-state systems. A program with a single integer variable x that is initially 0 and is repeatedly incremented by one, yields a nonterminating least fixed point computation. Widening techniques [CC77] are needed to accelerate the fixed point computation so that it does terminate with a fixed point that is not necessarily the least one.

A different, more conservative approach to invariant generation is given by the computation of the greatest fixed point of the strongest postcondition $\nu X.sp(P)(X)$. For example, a greatest fixed point computation on a program with a single variable x and a single guarded transition $x \geq 0 \longrightarrow x := x + 1$ would terminate and yield the invariant $x \geq 0$. The greatest fixed point invariant computation also may not terminate and could require *narrowing* as a way of accelerating termination. However, one could stop the greatest fixed point computation after any bounded number of iterations and the resulting predicate would always be a valid invariant.

Dually, a putative invariant p can be strengthened to an inductive one by computing the greatest fixed point with respect to the weakest precondition of the program of the given invariant $\nu X.p \wedge wp(P)(X)$. If this computation terminates, the result is an invariant that is inductive.

Automatic invariant generation is not yet a successful technology. Right now, it is best used for propagating invariants that are computed from other sources by taking the greatest fixed point with respect to the strongest post-condition starting from a known invariant. However, as theorem proving technology becomes more powerful and efficient, invariant generation is likely to be quite a fruitful technique.

2 Symbolic Analysis

Symbolic analysis is simply the computation of fixed point properties of programs through a combination of deductive and explorative techniques. We have already seen the key elements of symbolic analysis as

1. *Automated deduction*, in computing property preserving abstractions and propagating the consequences of known properties.
2. *Model checking*, as a means of computing global properties of by means of systematic symbolic exploration. For this purpose, model checking is used for actually computing fixed points such as the reachable state set, in addition to verifying given temporal properties.
3. *Invariant generation*, as a technique for computing useful properties and propagating known properties.

2.1 SAL: A Symbolic Analysis Laboratory

SAL is a framework for integrating different symbolic analysis techniques including theorem proving and model checking. The core of SAL is a description language for transition systems. The design of this intermediate language has been influenced by SMV [McM93], UNITY [CM88], Murphi [MD93], and Reactive Modules [AH96]. Transition systems described in SAL consist of modules with input, output, global, and local variables. Initializations and transitions can be either specified by definitions of the form $variable = expression$ or by guarded commands. The assignment part of a guarded command consists of assignments of the form $x' = expression$, meaning the new value of x is the value of the $expression$, as well as selections $x' \in set$, meaning the new value of x is nondeterministically selected from the value of the nonempty set set. SAL is a synchronous language in the spirit of Esterel [BG92], Lustre [HCRP91], and Reactive Modules [AH96], in the sense that transitions can depend on latched values as well as current inputs. SAL modules can be composed by means of

1. Binary synchronous composition $P||Q$ whose transitions consist of lock-step parallel transitions of P and Q.
2. Binary asynchronous composition $P\,[]\,Q$ whose transitions are the interleaving of those of P and Q.
3. N-fold synchronous composition $(||\ (i):\ P[i])$
4. N-fold asynchronous composition $([]\ (i):\ P[i])$

The implementation of SAL is still ongoing. The version to be released some time in 2000 will consist of a parser, typechecker, translators to SMV and PVS, a translator to Java (for animation), and a translator from Verilog, among other tools.

Since the SAL implementation is still incomplete, we informally describe some examples that motivate the need for a symbolic analysis framework integrating abstraction, invariant generation, theorem proving, and model checking.

2.2 Analysis of a Two Process Mutual Exclusion Algorithm

As a first example, we use a simplified 2-process version of Lamport's Bakery algorithm for mutual exclusion [Lam74]. The algorithm consists of two processes P and Q with control variables pcp and pcq, respectively, and shared variables

x and y. The control states of these processes are either `sleeping`, `trying`, or `critical`. Initially, *pcp* and *pcq* are both set to `sleeping` and the control variables satisfy $x = y = 0$. The transitions for P are

$$\begin{array}{lr}
 & \mathtt{pcp} = \mathit{sleeping} \longrightarrow x' = y + 1;\ \mathit{pcp}' = \mathtt{trying} \\
[\,]\ \mathit{pcp} = \mathtt{trying} \wedge (y = 0 \vee x < y) & \longrightarrow \mathit{pcp}' = \mathtt{critical} \\
[\,] & \mathit{pcp} = \mathtt{critical} \longrightarrow x' = 0;\ \mathtt{pcp}' = \mathtt{sleeping}
\end{array}$$

Similarly, the transitions for Q are

$$\begin{array}{lr}
 & \mathit{pcq} = \mathtt{sleeping} \longrightarrow y' = x + 1;\ \mathit{pcq}' = \mathtt{trying} \\
[\,]\ \mathit{pcq} = \mathtt{trying} \wedge (x = 0 \vee y \leq x) & \longrightarrow \mathit{pc}' = \mathtt{critical} \\
[\,] & \mathit{pcq} = \mathtt{critical} \longrightarrow y' = 0;\ \mathit{pcq}' = \mathtt{sleeping}
\end{array}$$

The invariant we wish to establish for $P\,[\,]\,Q$ is $\neg(\mathit{pcp} = \mathtt{critical} \wedge \mathit{pcq} = \mathtt{critical}$. Note that $P\,[\,]\,Q$ is an infinite-state system and in fact the values of the variables x and y can increase without bound. We can therefore attempt to verify the invariant by means of a property-preserving predicate abstraction to a finite-state system.

The abstraction predicates suggest themselves from the initializations, guards, and assignments. We therefore abstract the predicate $x = 0$ with the boolean variable x_0, the predicate $y = 0$ with the boolean variable y_0, and the predicate $x < y$ with the boolean variable xy. The resulting abstract system can be computed as P' and Q', where in the initial state, $x_0 \wedge y_0 \wedge \neg xy$, and the transitions for P' are

$$\begin{array}{lr}
 & \mathit{pcp} = \mathit{sleeping} \longrightarrow \mathrm{x}_0' = \mathbf{false};\ xy' = \mathbf{false};\ \mathit{pcp}' = \mathtt{trying}; \\
[\,]\ \mathit{pcp} = \mathtt{trying} \wedge (y_0 \vee xy) & \longrightarrow \mathit{pcp}' = \mathtt{critical}; \\
[\,] & \mathit{pcp} = \mathtt{critical} \longrightarrow x_0' = \mathbf{true}; xy' \in \{\mathbf{true}, \mathbf{false}\};\ \mathit{pcp}' = \mathtt{sleeping};
\end{array}$$

The transitions for Q' are

$$\begin{array}{lr}
 & \mathit{pcq} = \mathtt{sleeping} \longrightarrow y_0' = \mathbf{false};\ xy' = \mathbf{true};\ \mathit{pcq}' = \mathtt{trying}; \\
[\,]\ \mathit{pcq} = \mathtt{trying} \wedge (x_0 \vee \neg xy) & \longrightarrow \mathit{pcp}' = \mathtt{critical}; \\
[\,] & \mathit{pcp} = \mathtt{critical} \longrightarrow y_0' = \mathbf{true}; xy' = \mathbf{false};\ \mathit{pcp}' = \mathtt{sleeping};
\end{array}$$

Model checking the abstract system $P'\,[\,]\,Q'$ easily verifies the invariant

$$\neg(\mathit{pcp} = \mathtt{critical} \wedge \mathit{pcq} = \mathtt{critical}).$$

The theorem proving needed to construct the abstraction is at a trivial level that can be handled automatically by the decision procedures over quantifier-free formulas in a combination of theories [RS00]. Such decision procedures are present in systems like PVS [ORS92], ESC [Det96], SVC [BDL96], and STeP [MtSG95]. The above example can be verified fully automatically by means of the `abstract-and-model-check` command in PVS [SS99].

2.3 Analysis of an N-Process Mutual Exclusion Algorithm

We next examine a fictional example, namely, one that has not been mechanically verified by us. This example is a simplified form of the N-process Bakery algorithm due to Lamport [Lam74]. The description below shows a hand-executed symbolic analysis.

In this version of the Bakery algorithm, there are N processes $P(0)$ to $P(N-1)$, with a shared array x of size N over the natural numbers. The logical variables i, j, and k range over the subrange $0..(N-1)$. The operation $max(x)$ returns the maximal element in the array x. Initially, each $P(i)$ is in the control state `sleeping`, and for each i, $x(i) = 0$. Let $\langle x, i\rangle \leq \langle y, j\rangle$ be defined as the lexicographic ordering $x < y \vee (x = y \wedge i \leq j)$. We abbreviate $y = 0 \vee \langle x, i\rangle \leq \langle y, j\rangle$ as $\langle x, i\rangle \preceq \langle y, j\rangle$.

The transitions of processes $P(i)$ for $0 \leq i < N$ are interleaved and each non-stuttering transition executes one of the following guarded commands.

$$\begin{array}{lrl} & pc(i) = \mathit{sleeping} \longrightarrow & x'(i) = 1 + max(x); \\ & & pc'(i) = \texttt{trying}; \\ [] & pc(i) = \texttt{trying} \qquad\qquad \longrightarrow & pc'(i) = \texttt{critical}; \\ & \wedge\ (\forall j : \langle x(i), i\rangle \preceq \langle x(j), j\rangle) & \\ [] & pc(i) = \mathit{critical} \longrightarrow & x'(i) = 0; \\ & & pc'(i) = \texttt{sleeping}; \end{array}$$

We want to prove the invariance property

$$(\forall i : pc(i) = \texttt{critical} \supset (\forall j : pc(j) = \texttt{critical} \supset i = j)). \qquad (1)$$

Invariant generation techniques can be used to generate trivial invariants such as

$$(\forall i : x(i) = 0 \ \mathit{iff}\ pc(i) = \texttt{sleeping}). \qquad (2)$$

We omit the details of the invariant generation step. The above invariant will prove useful in the next stage of the analysis.

We next skolemize the mutual exclusion statement so as to obtain a correctness goal about a specific but arbitrary i which we call a. The main invariant now becomes

$$pc(a) = \texttt{critical} \supset (\forall j : pc(j) = \texttt{critical} \supset a = j) \qquad (3)$$

The goal now is to reduce the N-process protocol to a two process protocol consisting of process a and another process b that is an *existential abstraction* of the remaining $N-1$ processes. By an existential abstraction, we mean one where the $N-1$ processes are represented by a single process b such that a transition by any of the $N-1$ processes is mapped to a corresponding transition of b. In such an abstraction, b is in control state `critical` if any one of the $N-1$ processes is critical. Otherwise, b is in control state `trying` if none of the $N-1$ processes is in the state `critical` and at least one of them is in its `trying` state. If none of the $N-1$ process is either `trying` or `critical`, then b is in its `sleeping` state.

By examining the predicates appearing in the initialization, guards, and the property, we can directly obtain the following abstraction predicates given by the function γ which maps abstract variables to the corresponding concrete predicates:

$$
\begin{aligned}
\gamma(pca) &= pc(a) \\
\gamma(pcb) &= \begin{array}[t]{ll} \mathit{if} & (\exists j : j \neq a \wedge pc(j) = \texttt{critical}) \\ \mathit{then} & \texttt{critical} \\ \mathit{elsif} & (\exists j : j \neq a \wedge pc(j) = \texttt{trying}) \\ \mathit{then} & \texttt{trying} \\ \mathit{else} & \texttt{sleeping} \end{array} \\
\gamma(xa_0) &= (x(a) = 0) \\
\gamma(xb_0) &= (\forall j : j \neq a \supset x(j) = 0) \\
\gamma(ma) &= (\forall j : \langle x(a), a\rangle \preceq \langle x(j), j\rangle) \\
\gamma(mb) &= (\exists j : (\forall k : \langle x(j), j\rangle \preceq \langle x(k), k\rangle) \\
\gamma(ea) &= (\forall j : pc(j) = \texttt{critical} \supset a = j)
\end{aligned}
$$

Since mb is only relevant when $pc(j) = \mathit{trying}$ for $j \neq a$, we can use invariant (2) to prove that

$$j \neq a \wedge pc(j) \neq \mathit{sleeping} \supset \gamma(mb) = \gamma(\neg ma)$$

thereby dispensing with mb in the abstraction.

With the above abstraction mapping, the goal invariant (3) becomes

$$pca = \texttt{critical} \supset ea.$$

and the resulting abstracted transition system is one where initially

$$pca = \mathit{sleeping} \wedge pcb = \mathit{sleeping} \wedge xa_0 \wedge xb_0 \wedge ma \wedge ea$$

Each non-stuttering step in the computation of the abstract program executes one of the guarded commands shown in Figure 1.

Model checking the abstract protocol fails to verify the invariant

$$pca = \mathit{critical} \supset ea$$

as the model checker could generate the following counterexample sequence of transitions:

transition	*pca*	*xa*	*ma*	*ea*	*pcb*	*xb*
initially	*sleeping*	*true*	*true*	*true*	*sleeping*	*true*
3	*sleeping*	*true*	*false*	*true*	*trying*	*false*
4	*sleeping*	*true*	*false*	*false*	*critical*	*false*
1	*trying*	*false*	*false*	*false*	*critical*	*false*
8	*trying*	*false*	*true*	*false*	*critical*	*false*
2	*critical*	*false*	*true*	*false*	*critical*	*false*

$$
\begin{array}{lrl}
 & pca = \texttt{sleeping} \longrightarrow & xa' = \textbf{false}; \\
 & & ma' = xb; \\
 & & pca' = \texttt{trying}; \\
[] & pca = trying \wedge ma \longrightarrow & pca' = \texttt{critical}; \\
[] & pca = \texttt{critical} \longrightarrow & pca' = \texttt{sleeping}; \\
 & & ma' = xb; \\
 & & ea' = \neg(pcb = \texttt{critical}); \\
 & & xa' = \textbf{true}; \\
[] & pcb = \texttt{sleeping} \longrightarrow & pcb' = \texttt{trying};\ xb' = \textbf{false}; \\
 & & ma' = \neg xa \\
[] & pcb = \texttt{trying} \wedge \neg ma \longrightarrow & pcb' = \texttt{critical};\ ea' = \textbf{false}; \\
[] & pcb = \texttt{critical} \longrightarrow & pcb' = \texttt{sleeping}; \\
 & & ea' = \textbf{true}; \\
 & & ma' = \textbf{true}; \\
 & & xb' = \textbf{true}; \\
[] & pcb = \texttt{critical} \longrightarrow & pcb' = \texttt{trying}; \\
 & & ea' = true; \\
 & & ma' \in \{\textbf{true}, ma\}; \\
[] & pcb = \texttt{critical} \longrightarrow & ma' \in \{\textbf{true}, ma\};
\end{array}
$$

Fig. 1. Abstract transitions for the N-process Bakery Algorithm

An inspection of the counterexample and the abstract model confirms that the mutual exclusion invariant would follow if the invariant $\neg xa \wedge ma \supset ea$ were to hold. Mapped back in the concrete domain, this corresponds to

$$\forall i : x(i) \neq 0 \wedge (\forall j : x(j) = 0 \vee \langle x(i), i\rangle \leq \langle x(j), j\rangle) \supset (\forall j : pc(j) = \texttt{critical} \supset i = j).$$

This goal can be generalized as

$$(\forall i, j : x(i) \neq 0 \wedge (x(j) = 0 \vee \langle x(i), i\rangle) \leq \langle x(j), j\rangle \supset (pc(j) = \texttt{critical} \supset i = j)).$$

and further rearranged as

$$(\forall i, j : pc(j) = \texttt{critical} \supset (x(i) \neq 0 \wedge (x(j) = 0 \vee \langle x(i), i\rangle \leq \langle x(j), j\rangle)) \supset i = j).$$

By the invariant (2), we can eliminate the subformula $x(j) = 0$ and simplify the goal to the equivalent formula

$$(\forall i, j : pc(j) = critical \supset x(i) = 0 \vee \langle x(j), j\rangle \leq \langle x(i), i\rangle).$$

This can be rearranged as

$$(\forall j : pc(j) = critical \supset (\forall i : x(i) = 0 \vee \langle x(j), j\rangle \leq \langle x(i), i\rangle)).$$

But this is the just the invariant $pca = critical \supset ma$ which is already implied by the abstract model.

The safety property is thus verified by using a judicious combination of a small amount of theorem proving and model checking. The abstractions were suggested by the predicates in the text of the program. Simple invariant generation methods were adequate for generating trivial invariants. Theorem proving in the context of these invariants could be used to discharge the proof obligations needed to construct an accurate abstraction of the N-process protocol. Abstraction mappings of this sort are quite standard and work for many mutual exclusion and cache consistency algorithms [Sha97]. The abstract model did not discharge the main safety invariant but it was easy to extract the minimal condition needed to verify the invariant from the abstract model. A reachability analysis of the abstract model delivered enough useful invariants so that a small amount of theorem proving could discharge this condition. Neither the model checking nor the theorem proving used here is especially difficult. While some guidance is needed in selecting lemmas and conjectures, the proofs of these can be carried out with substantial automation.

3 Conclusion

We have argued that verification technology is best employed as an analysis technique to generate properties of specifications and programs rather than as a method for establishing the correctness of specific properties. Such a symbolic analysis framework can employ both theorem proving and model checking as appropriate to generate useful abstractions and automatically derive system properties.

Many ideas remain to be explored within the symbolic analysis framework. The construction of the symbolic analysis laboratory SAL as an open framework will support the exploration of ideas at the interface of theorem proving and model checking.

Acknowledgments. Many collaborators and colleagues have contributed ideas and code to the SAL language and framework, including Saddek Bensalem, David Dill, Tom Henzinger, Luca de Alfaro, Vijay Ganesh, Yassine Lakhnech, Cesar Muñoz, Sam Owre, Harald Rueß, John Rushby, Vlad Rusu, Hassen Saïdi, Eli Singerman, Mandayam Srivas, Jens Skakkebæk, and Ashish Tiwari. John Rushby read an earlier draft of the paper and suggested numerous improvements.

References

[AH96] Rajeev Alur and Thomas A. Henzinger. Reactive modules. In *Proceedings, 11th Annual IEEE Symposium on Logic in Computer Science*, pages 207–218, New Brunswick, New Jersey, 27–30 July 1996. IEEE Computer Society Press.

[BBLS92] Saddek Bensalem, Ahmed Bouajjani, Claire Loiseaux, and Joseph Sifakis. Property preserving simulations. In *Computer-Aided Verification, CAV '92*, volume 630 of *Lecture Notes in Computer Science*, pages 260–273, Montréal, Canada, June 1992. Springer-Verlag. Extended version available with title "Property Preserving Abstractions.".

[BBM97] Nikolaj Bjørner, I. Anca Browne, and Zohar Manna. Automatic generation of invariants and intermediate assertions. *Theoretical Computer Science*, 173(1):49–87, 1997.

[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, June 1992.

[BDL96] Clark Barrett, David Dill, and Jeremy Levitt. Validity checking for combinations of theories with equality. In Mandayam Srivas and Albert Camilleri, editors, *Formal Methods in Computer-Aided Design (FMCAD '96)*, volume 1166 of *Lecture Notes in Computer Science*, pages 187–201, Palo Alto, CA, November 1996. Springer-Verlag.

[BG92] G. Berry and G. Gonthier. The Esterel synchronous programming language: Design, semantics, and implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[BL99] Saddek Bensalem and Yassine Lakhnech. Automatic generation of invariants. *Formal Methods in Systems Design*, 15(1):75–92, July 1999.

[BLO98] Saddek Bensalem, Yassine Lakhnech, and Sam Owre. Computing abstractions of infinite state systems compositionally and automatically. In Hu and Vardi [HV98], pages 319–331.

[BLS96] Saddek Bensalem, Yassine Lakhnech, and Hassen Saïdi. Powerful techniques for the automatic generation of invariants. In Rajeev Alur and Thomas A. Henzinger, editors, *Computer-Aided Verification, CAV '96*, volume 1102 of *Lecture Notes in Computer Science*, pages 323–335, New Brunswick, NJ, July/August 1996. Springer-Verlag.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis. In *4th ACM Symposium on Principles of Programming Languages*. Association for Computing Machinery, January 1977.

[CGL94] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, September 1994.

[CGP99] E. M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, 1999.

[CH78] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables. In *5th ACM Symposium on Principles of Programming Languages*. Association for Computing Machinery, January 1978.

[CM88] K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, Reading, MA, 1988.

[CU98] M. A. Col on and T. E. Uribe. Generating finite-state abstractions of reactive systems using decidion procedures. In Hu and Vardi [HV98], pages 293–304.

[Dam96] Dennis René Dams. *Abstract Interpretation and Partition Refinement for Model Checking*. PhD thesis, Eindhoven University of Technology, P.O. Box 513, 5600 MB Eindhoven, The Netherlands, July 1996.

[DDP99] Satyaki Das, David L. Dill, and Seungjoon Park. Experience with predicate abstraction. In Halbwachs and Peled [HP99], pages 160–171.

[Det96] David L. Detlefs. An overview of the Extended Static Checking system. In *First Workshop on Formal Methods in Software Practice (FMSP '96)*, pages 1–9, San Diego, CA, January 1996. Association for Computing Machinery.

[DF95] Jürgen Dingel and Thomas Filkorn. Model checking for infinite state systems using data abstraction, assumption-commitment style reasoning and theorem proving. In *Computer-Aided Verification 95*, 1995. This volume.

[DGG94] Dennis Dams, Orna Grumberg, and Rob Gerth. Abstract interpretation of reactive systems: Abstractions preserving ∀CTL*, ∃CTL* and CTL*. In Ernst-Rüdiger Olderog, editor, *Programming Concepts, Methods and Calculi (PROCOMET '94)*, pages 561–581, 1994.

[GM93] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: A Theorem Proving Environment for Higher-Order Logic*. Cambridge University Press, Cambridge, UK, 1993.

[GW75] S. M. German and B. Wegbreit. A synthesizer for inductive assertions. *IEEE Transactions on Software Engineering*, 1(1):68–75, March 1975.

[HCRP91] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language Lustre. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[HP99] Nicolas Halbwachs and Doron Peled, editors. *Computer-Aided Verification, CAV '99*, volume 1633 of *Lecture Notes in Computer Science*, Trento, Italy, July 1999. Springer-Verlag.

[HSV94] L. Helmink, M. P. A. Sellink, and F. W. Vaandrager. Proof-checking a data link protocol. Technical Report CS-R9420, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, The Netherlands, March 1994.

[HV98] Alan J. Hu and Moshe Y. Vardi, editors. *Computer-Aided Verification, CAV '98*, volume 1427 of *Lecture Notes in Computer Science*, Vancouver, Canada, June 1998. Springer-Verlag.

[Jan93] G. Janssen. *ROBDD Software*. Department of Electrical Engineering, Eindhoven University of Technology, October 1993.

[JS93] Jeffrey J. Joyce and Carl-Johan H. Seger. Linking BDD-based symbolic evaluation to interactive theorem proving. In *Proceedings of the 30th Design Automation Conference*. Association for Computing Machinery, 1993.

[KM76] S. Katz and Z. Manna. Logical analysis of programs. *Communications of the ACM*, 19(4):188–206, 1976.

[Kur94] R. P. Kurshan. *Computer-Aided Verification of Coordinating Processes—The Automata-Theoretic Approach*. Princeton University Press, Princeton, NJ, 1994.

[Lam74] Leslie Lamport. A new solution of Dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8):453–455, August 1974.

[LGS+95] C. Loiseaux, S. Graf, J. Sifakis, A. Bouajjani, and S. Bensalem. Property preserving abstractions for the verification of concurrent systems. *Formal Methods in System Design*, 6:11–44, 1995.

[MBSU99] Zohar Manna, Anca Browne, Henny B. Sipma, and Tomás E. Uribe. Visual abstractions for temporal verification. In Armando M. Haeberer, editor, *Algebraic Methodology and Software Technology, AMAST'98*, volume 1548 of *Lecture Notes in Computer Science*, pages 28–41, Amazonia, Brazil, January 1999. Springer-Verlag.

[McM93] Kenneth L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Boston, MA, 1993.

[McM99] K. L. McMillan. Verification of infinite state systems by compositional model checking. In Laurence Pierre and Thomas Kropf, editors, *Correct Hardware Design and Verification Methods*, number 1703 in Lecture Notes in Computer Science, pages 219–233. Springer Verlag, September 1999.

[MD93] Ralph Melton and David L. Dill. *Murφ Annotated Reference Manual*. Computer Science Department, Stanford University, Stanford, CA, March 1993.

[MQS00] K. McMillan, S. Qadeer, and J. Saxe. Induction in compositional model checking. In E. A. Emerson and A. P. Sistla, editors, *Computer-Aided Verification*, Lecture Notes in Computer Science. Springer Verlag, 2000. To appear.

[MtSG95] Z. Manna and the STeP Group. STeP: The Stanford Temporal Prover. In Peter D. Mosses, Mogens Nielsen, and Michael I. Schwartzbach, editors, *TAPSOFT '95: Theory and Practice of Software Development*, volume 915 of *Lecture Notes in Computer Science*, pages 793–794, Aarhus, Denmark, May 1995. Springer Verlag.

[ORS92] Sam Owre, John M. Rushby, and Natarajan Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *Automated Deduction - CADE-11, 11th International Conference on Automated Deduction, Lecture Notes in Artifical Intelligence*, pages 748–752. Springer Verlag, June 1992.

[Par76] David Park. Finiteness is mu-ineffable. *Theoretical Computer Science*, 3:173–181, 1976.

[RS00] H. Rueß and N. Shankar. Deconstructing Shostak. Available from http://www.csl.sri.com/shankar/shostak2000.ps.gz., January 2000.

[RSS95] S. Rajan, N. Shankar, and M.K. Srivas. An integration of model-checking with automated proof checking. In Pierre Wolper, editor, *Computer-Aided Verification, CAV '95*, volume 939 of *Lecture Notes in Computer Science*, pages 84–97, Liege, Belgium, June 1995. Springer-Verlag.

[Saï96] Hassen Saïdi. A tool for proving invariance properties of concurrent systems automatically. In *Tools and Algorithms for the Construction and Analysis of Systems TACAS '96*, volume 1055 of *Lecture Notes in Computer Science*, pages 412–416, Passau, Germany, March 1996. Springer-Verlag.

[Seg98] Carl-Johan H. Seger. Formal methods in CAD from an industrial perspective. In Ganesh Gopalakrishnan and Phillip Windley, editors, *Formal Methods in Computer-Aided Design (FMCAD '98)*, volume 1522 of *Lecture Notes in Computer Science*, Palo Alto, CA, November 1998. Springer-Verlag.

[SG97] Hassen Saïdi and Susanne Graf. Construction of abstract state graphs with PVS. In Orna Grumberg, editor, *Computer-Aided Verification, CAV '97*, volume 1254 of *Lecture Notes in Computer Science*, pages 72–83, Haifa, Israel, June 1997. Springer-Verlag.

[Sha97] N. Shankar. Machine-assisted verification using theorem proving and model checking. In M. Broy and Birgit Schieder, editors, *Mathematical Methods in Program Development*, volume 158 of *NATO ASI Series F: Computer and Systems Science*, pages 499–528. Springer, 1997.

[SI77] N. Suzuki and K. Ishihata. Implementation of an array bound checker. In *4th ACM Symposium on Principles of Programming Languages*, pages 132–143, January 1977.

[SS99] Hassen Saïdi and N. Shankar. Abstract and model check while you prove. In Halbwachs and Peled [HP99], pages 443–454.