An Integration of Model Checking with Automated Proof Checking*

Short version to appear in the proceedings of CAV '95

S. Rajan, N. Shankar, and M.K. Srivas
Computer Science Laboratory
SRI International
Menlo Park CA 94025 USA
{sree, shankar, srivas}@csl.sri.com
Phone: +1 (415) 859-5272 Fax: +1 (415) 859-2844

May 18, 1995

Abstract

Although automated proof checking tools for general-purpose logics have been successfully employed in the verification of digital systems, there are inherent limits to the efficient automation of expressive logics. If the expressiveness is constrained, there are useful logic fragments for which efficient decision procedures can be found. The model checking paradigm yields an important class of decision procedures for establishing temporal properties of finite-state systems. Model checking is remarkably effective for automatically verifying finite automata with relatively small state spaces, but is inadequate when the state spaces are either too large or unbounded. For this reason, it is useful to integrate the complementary technologies of model checking and proof checking. Such an integration has to be carried out in a delicate manner in order to be more than just the sum of the techniques. We describe

*Supported by ARPA under contract PR8556, by NSF Grant CCR-930044, and by NASA under contract NAS1-20334. We thank John Rushby (SRI) for encouraging us to write this paper and providing comments on earlier drafts of the paper. We are grateful to Geert Janssen (Eindhoven University of Technology) for supplying us the BDD-based mu-calculus model checker used in this work and going well beyond the call of duty in responding to our queries.

an approach for such an integration where a BDD-based model checker for the propositional mu-calculus has been used as a decision procedure within the framework of the PVS proof checker. We argue that our approach fits in nicely with the design philosophy of PVS of providing highly effective mechanical reasoning capability by using efficient decision procedures as the workhorses of an interactive proof checker.

1 Introduction

In the theorem proving approach to verification, a system and its properties are described by means of logical formulae and the system is shown by means of a logical proof to entail the desired properties. This approach when supplemented with the use of automated theorem proving tools has been employed successfully in the verification of digital hardware and software systems. Though this approach is very general and applies to a large variety of systems and properties, there are inherent limits to the efficiency with which expressive general-purpose logics can be fully mechanized. There are two approaches for dealing with this limitation. The first approach is to use interactive proof checkers so that correctness proofs can be developed using a combination of user guidance and limited forms of automated deduction. The second approach has been to find useful fragments of logic that can be mechanized very effectively. Model checking is an important instance of the latter approach for the verification of temporal properties of finite-state systems. The finite-state system is represented as a Kripke model and the system property is represented as a formula in propositional temporal logic. For certain temporal logics such as CTL [5], the model-checking problem is linear even when the decidability of the logic itself is EXPTIME-complete [9].

Model checking thus provides a fully automatic technique for deciding an important class of verification problems. The importance of such automation cannot be overemphasized. The effort required to construct logical proofs of correctness for these problems can be large since it requires the discovery of suitably strong invariant. The primary disadvantage of model checking is that it only works well for small state spaces. This limitation can be partially overcome through the use of binary decision diagrams (BDD) and symbolic model checking [3]. Here the state space and the automaton transition relation are represented by means of binary decision diagrams which provide a representation for entire classes of states rather than individual states. Even so, the state explosion problem limits the applicability of model checking in practice. BDDs also require a lot of careful attention to variable ordering which often requires significant manual effort. A lot of careful reasoning is required to abstract the real problem into one with a finite state.

Since proof checking and model checking are complementary technologies, it seems reasonable to somehow combine them. Even so, there has been very little progress in achieving such a combination in an effective manner. The HOL/Voss system [15] is an early attempt in this direction. In this combination, the HOL proof checker [12] is given input that contains constants that are uninterpreted for HOL but given an interpretation in Voss which is a symbolic model checker. Voss is used to establish properties of these constants and the resulting assertions are fed back to the HOL proof as lemmas. In the HOL/Voss implementation the connection between the two systems is not close enough for the properties established by Voss to be proved directly in the unextended HOL system.

Kurshan and Lamport [17] present a similar connection between the TLP proof checker for TLA, the temporal logic of actions, and COSPAN [18] which is a language containment verifier based on BDDs. They present a proof of a multiplier where the 8-bit multiplier can be verified by COSPAN and N-bit multiplier composed from 8-bit multipliers can be verified in TLP [11]. In [17], the two systems used to verify the multiplier were used independently and not actually integrated. Hungar describes a similar effort where a model checker is used to verify properties of processes and a syntactic formalization of MCTL is used to verify the composition of the individual processes.

There is other relevant work where model checking has been extended in significant ways to deal with problems involving large and unbounded state spaces. Wolper and Lovinfosse [25] and Kurshan and McMillan [19] present techniques where model checking can be used for the inductive step by using an invariant to capture the induction hypothesis. This approach can be used for example to verify an N-bit buffer and the N-dining philosopher problem.

The goal of the present work is to present a smooth and tight integration of model checking with theorem proving in the context of the PVS proof checker [23]. The propositional mu-calculus serves as a basis of our approach. An extension of the mu-calculus is defined using the higher-order logic of PVS. The temporal operators are then given their customary fixpoint definitions using the mu-calculus. These temporal operators apply to arbitrary state spaces. In the instance when the state type is constructed in a hereditarily finite manner, we translate mu-calculus expressions to input acceptable by a mu-calculus model checker. This model checker can then be used as a decision procedure within a proof to prove certain subgoals. This yields a tight integration between PVS and the mu-calculus model checker since the latter is used as a decision procedure for a well-defined fragment of PVS.

Typical uses for the capability provided by our integrated system include the verification of temporal properties of finite-state abstractions of systems, and the use of model checking in the induction step of iterated finite-state systems. Examples verified by the combined system include Peterson's 2-process mutual exclusion algorithm, the alternating-bit protocol, a simple counter, and an arbiter for a CPU running concurrently with a lookahead-fetch unit. Our basic approach can be generalized to also integrate temporal logic model checkers such as SMV [20] and language containment systems such as COSPAN [18].

Our approach is quite similar to that of Dingel and Filkorn [8] in that they use a combination of a model checker for LTL and a theorem prover for first-order logic, but with a looser integration than the one we present. Müller and Nipkow [22] also describe a combination of model checking and theorem proving using I/O automata where they verify the main safety property of the alternating bit protocol using a property-preserving abstraction that yields a finite-state system. Such property-preserving abstractions in the context of model checking have also been studied by Graf [13], Clarke, Grumberg, and Long [5], and Dams, Grumberg, and Gerth [7].

The rest of the paper is organized as follows: The propositional mu-calculus is introduced in Section 2.1 The definition of CTL and fairCTL operators in terms of mu-calculus and their embedding in PVS are discussed in Section 2.2. The translation of a finite-state fragment of the mu-calculus into input acceptable by a model checker for propositional mu-calculus is shown in Section 2.3. The use of abstraction for reducing the verification of arbitrary transition system to finite-state ones is described in Section 3. Section 4 presents the basic methodology for using the model checker as a decision procedure within a proof, and then describe a significant example that exploits the model checker. This example was motivated by a problem that we encountered during the verification of a commercial processor, AAMP5 [21], using PVS. The verification of AAMP5, which contains almost half a million transistors, is too large to be verified entirely by a model checker. Conclusions are summarized in Section 5.

2 Model Checking within Theorem Proving

Our primary design choice in integrating model checking with PVS is to view model checking as a decision procedure for a well-defined fragment of PVS. Transition systems can be described in terms of a next-state relation over a specific state type. The temporal operators then permit assertions over the states and computation paths in such a transition system. The mu-calculus can be used to define such temporal operators. The higher-order logic of PVS is used to define a mu-calculus theory that is parametric in its state type. The CTL operators can be defined using the mu-calculus. These operators are parametric both in the state type and a given next-state relation over this state type. Formulas in the mu-calculus over finite state types can be translated into the propositional mu-calculus where the state type is just a tuple consisting of booleans. We first present the mu-calculus and the propositional mu-calculus. We then briefly describe the definition of the CTL and fairCTL operators. Finally, we discuss the translation between the mu-calculus over finite types and the propositional mu-calculus so that a decision procedure for the propositional case can be used for the finite case.

2.1 Propositional mu-calculus and Temporal Logic: Overview

Propositional mu-calculus is an extension of propositional calculus that includes universal and existential quantification on propositional variables (i.e., quantified Boolean formulas), and predicates defined by means of the least and greatest fixpoint operators, μ and ν, respectively. It is strictly more expressive than CTL*, and provides a framework to express fairness and extended temporal modalities [10].

There have been several variations of mu-calculus proposed in the past [3, 6, 10, 16, 24]. We closely follow the formal definition of the syntax of propositional mu-calculus from Burch, et al [3], that forms the basis of the model checker [14] used in this work. Let Σ be a finite signature, in which every symbol is a propositional variable or a predicate variable with a positive arity. The two syntactic categories *formulas* and *relational terms* are defined in the following manner. A formula is either:

- a propositional variable z in Σ

- the conjunction, negation, disjunction, implication, or equivalence of formulas: $f \wedge g$, $\neg f$, $f \vee g$, $f \supset g$, or $f = g$
- an n-ary relational term p applied to a list of n formulas $f_1, \ldots, f_n$: $p(f_1, \ldots, f_n)$
- the result of apply existential or universal quantification of a variable over a formula: $\exists z.\ f$ or $\forall z.\ f$

A n-ary relational term is one of the following:

- Z, an n-ary predicate variable in Σ
- $\lambda z_1, z_2, \ldots, z_n.\ f$, where f is a formula and $z_1, z_2, \ldots, z_n$ are propositional variables in Σ.
- $\mu Z.\ P[Z]$, denoting the least fixpoint of P. Here, Z is an n-ary predicate variable in Σ and $P[Z]$ is a relational term formally monotonic in Z (i.e., Z occurs under an even number of negations in $P[Z]$). Similarly, $\nu Z.\ P$ is the greatest fixpoint of P, and is equivalent to the negation of the least fixpoint of $\neg P[\neg Z]$.

The satisfiability and model-checking problems for propositional mu-calculus expressions are decidable since fixpoints exist and can be computed. We can generalize the propositional mu-calculus to obtain a mu-calculus for an arbitary type by allowing relational terms to be predicates over this *state* type. Quantification must also be generalized to range over the state type. Stirling and Bradfield [1] describe a tableau proof system for a similar mu-calculus.

Temporal logics such as CTL with extensions of fairness (fairCTL) and other temporal modalities can be succinctly expressed using the mu-calculus [3, 10] defined above. Additionally, it has been shown that LTL model checking can be reduced to fairCTL model checking [4]. CTL is a branching-time temporal logic that can quantify over paths in a computation tree. It can thus capture temporal *possibility* but not, in general, *inevitability*. The latter notion requires a linear-time temporal logic. The definitions of selected CTL operators are shown below. Let N be a binary *next-state* relation over the state type σ and let p and q be relational terms over σ. The predicate $(\mathbf{EX}p)$ holds at a state x if p holds at some successor state. The predicate $\mathbf{EG}p$ holds at x if p holds on every state along some infinite path of successive states leading out of x. The predicate $\mathbf{E}(p\mathbf{U}q)$ holds at state x if there is a state y where predicate q holds that is reachable along a path of successive states leading out of x where p holds until q does.

$$\begin{array}{rcl}
(\mathbf{EX}p)(x) & = & \exists z.\ p(z) \wedge N(x,z) \\
(\mathbf{EG}p)(x) & = & (\nu Z.\ (\lambda z.\ p(z) \wedge (\mathbf{EX}Z)))(x) \\
(\mathbf{E}(p\mathbf{U}q))(x) & = & (\mu Z.\lambda z.\ q(z) \vee (p(z) \wedge (\mathbf{EX}Z)(z)))(x)
\end{array}$$

2.2 Mu-Calculus and CTL in PVS

PVS employs a specification language based on a simply typed higher-order logic so that it is permissible to quantify over predicate variables and variables that are functions, or functions of functions, and so on. The type of functions from type `S` to `T` is represented as `[S -> T]`. The type of predicates over type T is then represented as `[T -> bool]` (abbreviated as `PRED[T]`), where `bool` is the type of booleans consisting of `TRUE` and `FALSE`. The everywhere-**true** predicates $\top$ and the everywhere-**false** predicate $\bot$ can be represented as `LAMBDA (x:T): TRUE` and `LAMBDA (x:T): FALSE`, respectively. We can define a pointwise ordering `<=` of predicates, say `p1` and `p2`, of type `PRED[T]` as `(p1 <= p2) = (FORALL (x:T): p1(x) IMPLIES p2(x))`. When `p1 <= p2` we say that `p1` is stronger than `p2` or conversely that `p2` is weaker than `p1`.

We can lift the logical operations like conjunction, negation, and disjunction to predicates by overloading the symbols used for the corresponding boolean operations. For example, `p1 AND p2` can be defined as `(FORALL (x:T): p1(x) AND p2(x))`. A *predicate transformer* for predicates over `T` has the type `[PRED[T] -> PRED[T]]`. A predicate transformer is monotonic if it preserves the `<=` ordering on predicates. Given a monotonic predicate transformer `pp`, the predicate `mu(pp)` is defined to be the least fixpoint of `pp`, and `nu(pp)` is defined to be its greatest fixpoint. The Tarski-Knaster argument for the unique existence of these fixpoints is easily verified in PVS.

Given `mu` and `nu`, we can define the CTL operators so that they are parametric in the next-state relation `N` of type `[T, T -> bool]`, where `f`, `g`, and `h` range over predicates of the state type.

```
EX(N,f)(u):bool    =  (EXISTS v: (f(v) AND N(u, v)))
```

```
EG(N,f):PRED[T]    =  nu(LAMBDA Q: (f AND EX(N,Q)))

EU(N,f,g):PRED[T]  =  mu(LAMBDA Q: (g OR (f AND EX(N,Q))))
```

It is useful to be able to assert that a temporal property holds along some fair path or along all fair paths. There are many different notions of fairness. A simple and useful notion of fairness is given by characterizing the fair paths as those along which a fairness predicate holds infinitely often. This form of fairness cannot be expressed in CTL but can easily be defined in the mu-calculus. Let `fairEG(N, f)(h)(u)` assert the existence of a fair path from `u` along which `f` holds on each state and `h` holds infinitely often. This has the following definition in the PVS formalization of the mu-calculus.

```
fairEG(N, f)(h)  =  nu(LAMBDA p. EU(N, f, f AND h AND EX(N, p)))
```

The other fairCTL operators, `fairAF`, `fairAG`, `fairEF`, etc., are defined by using the `fairEG` operator in the same manner as Burch et al [2,20]. The advantage of having an explicit formalization of fairness in a verification system is that it allows one to check if there exists at least one fair path in a given model. Without such an explicit formalization, there is a danger of imposing fairness constraints that are never satisfied by the transition system so that many properties might hold trivially.

2.3 Translation from PVS to mu-calculus

Since the low-level BDD-based mu-calculus model checker accepts only the language of propositional mu-calculus as discussed in Section 2.1, an automatic translation is provided from the mu-calculus fragment of the PVS language to propositional mu-calculus.

The fragment of the PVS language that is translated into mu-calculus consists of expressions involving types that are finite, i.e., constructed inductively from the booleans or scalar types using type constructors that are either tuples, records, or arrays over a specific finite subrange of the integers. The type of booleans is written in PVS as `bool`. A scalar type consisting of $c_1, \ldots, c_n$ is written as $\{c_1, \ldots, c_n\}$. A subrange type from specific integers lo to hi is written as `subrange[`lo, hi`]`. A record consisting of n labels l_i of type T_i is written as `[#` $l_1 : T_1, \ldots, l_n : T_n$ `#]`. A tuple consisting of n

types $T_1, \ldots, T_n$ is written as $[\mathtt{T}_1, \ldots, \mathtt{T}_\mathtt{n}]$. An array of element type T over a specific subrange of the integers is written as `[subrange[`lo, hi`] ->` T`]`.

The details of the translation from PVS to the propositional mu-calculus are easily described by means of an example. Consider a state type `state` given by the following PVS declarations:

```
ACK : TYPE = {ready, wait}
DATA : TYPE = [subrange[0,1] -> bool]
state : TYPE = [# request: bool, ack: ACK, data : DATA #]
s, s1, s2 : VAR state
P, Q : VAR PRED[state]
i, j : VAR subrange[0, 1]
```

If we take the PVS formula

```
ack(s) = ready IMPLIES EU(N, (LAMBDA s1: ack(s) = ready),
                              (LAMBDA s1: NOT request(s1)))(s)
```

and expand the definition of `EU`, we obtain

```
ack(s) = ready IMPLIES
 mu(LAMBDA Q:
    (LAMBDA s1:
      NOT request(s1)
     OR (ack(s1) = ready
         AND
         (EXISTS s2:
          Q(s2) AND
          ((s2 = s1 WITH [request := NOT request(s1)])
           OR (request(s1) AND s2 = s1 WITH [ack := ready])
           OR (NOT request(s1) AND s2 = s1 WITH [ack := wait]))
            ))))(s)
```

The translation of the state `s` into the propositional mu-calculus is given by a tuple consisting of a boolean variable x_1 for `request(s)`, a boolean variable x_2 which is **false** if `ack(s)` is `ready`, and **true** if it is not, and two boolean variables x_3 and x_4 corresponding to `data(s)(0)` and `data(s)(1)`, respectively. The state variables `s1` and `s2` can similarly be encoded in terms of variables $y_1, \ldots, y_4$ and $z_1, \ldots, z_4$, respectively. Since the scalar type `ACK` is represented by a single boolean variable indicating `ready` when **false**, and `wait` when **true**, the PVS formula `ack(s) = ready` is simply translated as $\neg x_2$. The entire PVS formula above is therefore translated as

$$
\begin{array}{l}
\neg x_2 \\
\quad \supset \ (\mu Q.\ \lambda y_1, \ldots, y_4. \\
\qquad\qquad\qquad \neg y_1 \\
\qquad\qquad \vee \qquad \neg y_2 \\
\qquad\qquad\quad \wedge \quad \exists z1, \ldots, z_4. \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad Q(z_1, \ldots, z_n) \\
\qquad\qquad\qquad\qquad\quad \wedge \qquad\quad (\quad (z_1 = \neg y_1 \wedge z_2 = y_1 \wedge z_3 = y_3 \wedge z_4 = y_4)) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vee \quad (y_1 \wedge (z_1 = y_1 \wedge \neg z_2 \wedge z_3 = y_3 \wedge z_4 = y_4)) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vee \quad (\neg y_1 \wedge (z_1 = y_1 \wedge z_2 \wedge z_3 = y_3 \wedge z_4 = y_4))) \\
\qquad)(x_1, x_2, x_3, x_4)
\end{array}
$$

The above translation has been automated in PVS. There is a single atomic proof step in PVS that can take a goal given by a PVS formula containing `mu` and `nu` operators, translate this to the propositional mu-calculus as shown above, and apply BDD-based model checking to this formula. The application of the model checker either proves the goal, returns a list of one or more subgoals corresponding to collection of initial states where the property fails to model check, or it merely applies boolean simplification to the goal. We have defined a PVS proof strategy that carries out a sequence of inference steps that simplify goal formulas written in the CTL fragment of PVS by expanding out the definitions of the CTL operators in terms of the `mu` and `nu` operators, and applies the model checking proof step to the result.

The fragment of the PVS language given above is rich enough to express specifications and properties of state-machine models in a structured manner. In comparison to language front-ends for other model checkers such as SMV [20], the PVS sublanguage used for model checking is more expressive, and has the significant advantage of a proof system for the language.

3 Using Model Checking during Proof Checking

Using the model checker to verify CTL or any other mu-calculus property of a finite-state system is of course straightforward. The state of the system is presented as a finite type in PVS. The system is then described in terms of an initialization predicate and a next-state relation. System properties can be expressed in the CTL fragment or in terms of any other operators definable using the mu-calculus. Such properties can be proved by a single proof command called `model-check`. This usage of the model-checking capability in PVS is not much of an improvement over a conventional model checker. The real increase in convenience comes from the ability to combine model checking with the use of abstraction, induction, and compositionality. All of these techniques have been well studied in the model checking literature but the bulk of the reasoning is carried out informally. We illustrate how our combined technology can be applied to these problems.

The use of abstraction is fundamental to exploiting the combination of theorem proving and model checking. Many simple system properties are expressible in ∀CTL whose formulas in negation normal form, i.e., with only atomic negations, contain only the universal **A** path quantifier and not the existential **E** path quantifer. As shown by Clarke, Grumberg, and Long [5], there is a simple way to construct abstractions in this case. Given a concrete state type C and an abstract state type A, we need a surjective mapping h from C to A that "preserves" the initialisation predicate, the next-state relation, and the property of interest. Let the concrete transition system M_C be described in terms of an initialisation predicate I_C and a next-state relation N_C, and similarly the abstract transition system M_A is given in terms of I_A and N_A. In order to show $M_C \models p_C$ for a concrete state formula p_C written in ∀CTL or in the more expressive ∀CTL*, it is sufficient to prove[1]

- $\forall(c : C) : I_C(c) \supset I_A(h(c))$
- $\forall(a_1, a_2 : A) : (\exists(c_1, c_2 : C) : c_1 = h(a_1) \wedge c_2 = h(a_1) \wedge N_C(c_1, c_2)) \supset N_A(a_1, a_2)$
- $p_A \supset_h p_C$ holds, where $p_A \supset_h p_C$ is defined for the case of ∀CTL as:
  - $\mathbf{AG}p_A \supset_h \mathbf{AG}p_C$ iff $p_A \supset_h p_C$
  - $\mathbf{AF}p_A \supset_h \mathbf{AF}p_C$ iff $p_A \supset_h p_C$
  - $\mathbf{A}(p_A \mathbf{U} q_A) \supset_h \mathbf{A}(p_C \mathbf{U} q_C)$ iff $p_A \supset_h p_C$ and $q_A \supset_h q_C$

[1] These conditions are somewhat different from those given by Clarke, Grumberg, and Long [5].

- $\circ$ $(\forall (c : C) : p_A(h(c)) \supset p_C(c))$, when p_A and p_C contain no temporal operators.

- $M_A \models p_A$

A stronger version of the above conditions on abstraction is used in Section 4 to verify a liveness property of a simple pipelined microprocessor. These conditions on the abstraction can be extended in several ways to preserve properties in all of CTL or CTL* [5, 7]. Dams, Grumberg, and Gerth [7] present a notion of mixed abstraction that preserves all CTL* properties but involves multiple next-state relations. The mu-calculus can in fact capture temporal formulas involving multiple next-state relations.

4 Abstraction to Finite State

In this section, we demonstrate an application for our integrated facility where the model checker is used as a primitive step, i.e., a decision procedure, in a PVS proof. The application also illustrates the use of abstraction as a means of decomposing a potentially tedious manual proof into two automatic proofs, one involving theorem proving and the other model checking. The problem is a simplified version of a real verification problem that arose in the context of verifying a commercial microprocessor [21].

We verify a property of a small microprocessor CPU design that is an extension of the CPU example used by Burch, et al, [3] to illustrate the power of symbolic model checking. The example is a register-transfer level design of the datapath and controller of a microprocessor that executes instructions of the form `(opcode src1 src2 dstn)` to perform both register-register and register-memory operations. The CPU consists of a three stage read-execute-write pipeline with suitable control logic to handle external asynchronous (hand-shake) memory interaction and look-ahead instruction fetch. The only assumption made about the memory is that every read/write request by the design is eventually acknowledged (`ack`) and that the memory operation is correctly completed when an acknowledgement is received. The signal `ack` is required to become false one cycle after it becomes true and is implicitly assumed to be true infinitely often by virtue of the fact that we prove the correctness theorem under the fairness assumption of `ack` being true.

The CPU is *held* or frozen (indicated by `dhld`) till a data read/write to memory is acknowledged, and *stalled* (by introducing a stream of noop instructions) as long as the next instruction is not ready (i.e., `instrn_rdy` does not hold). We want to prove a property `next_instrn_entry` (shown below) that along all fair paths (where `ack` occurs infinitely often), if the next instruction (at the current `pc`) is not yet fetched, its opcode in the updated memory will eventually be loaded into the appropriate pipeline register, namely, `opcoded`. This loading could be delayed either because the machine is stalled or frozen. Hence a proof of this property relies on two lemmas characterizing the behavior of the CPU when it is either held or stalled. The `fetch_completes` lemma shows that the machine will eventually unstall (given the fairness condition) without changing the value of the `pc`. The second lemma shows that machine will eventually be unfrozen again without changing the value of the `pc`. The main theorem can be deduced from these lemmas using PVS.

```
next_instrn_entry: THEOREM
  NOT instrn_rdy(s0)
    => fairAF(N, λ (s1): instrn_rdy(s1) &
                 AX(N, λ (s2):
                         opcoded(s2) =
                         opcodeof(memory(s1)(pc(s0))))(s1))(ack)(s0)

fetch_completes: LEMMA
   NOT instrn_rdy(s0) IMPLIES
     fairAF(N, λ (s1): instrn_rdy(s1) & pc(s0) = pc(s1))(ack)(s0)

write_completes: LEMMA
   dhld(s0) IMPLIES
     fairAF(N, λ (s1): NOT dhld(s1) & pc(s1) =  pc(s0))(ack)(s0)
```

The above lemmas are not easy prove directly in PVS since they involve induction on the length of time for the memory to respond with an acknowledgment. Complicating the induction proof is the fact that the stalling loop and the data-holding loop are interdependent. Our approach is to abstract away the irrelevant parts of the processor state so that we are left with a finite-state processor-memory system that preserves the properties of interest. The relevant components in this case are: `ack`, the signals `write`, `next_write`, which determine `instrn_rdy` and `dhld`, `opcoded`, and `pc`. The state of the memory is completely abstracted except for its control signals.

Since we are only interested in analyzing whether the value of `pc` has changed from the initial state for the property, it is only necessary to retain a 1-bit information about the `pc` that indicates if the concrete program counter will be updated in the current state. The updating of pc in the abstract model captures the conditions under which the program counter is not changed in the concrete model. The abstraction function (shown below) takes an additional parameter init_pc_val that denotes the value with respect to which the concrete pc is compared to get the abstract pc value.

```
abs(init_pc_val: word)(cs): staterec =
  (# write := write(cs),
     next_write := next_write(cs),
     ack := ack(cs),
     pc := (pc(cs) = init_pc_val) #)

homo_morphic: LEMMA
     (abs_pipe.N(s1, s2)) IFF EXISTS (cs1, cs2):
          abs(pc(cs1))(cs1) = s1 & abs(pc(cs1))(cs2)= s2 &
                                   concrete_pipe.N(cs1, cs2)

congruent: LEMMA FORALL (wd: word): LET abs = abs(wd) IN
  instrn_rdy(abs(cs)) IFF instrn_rdy(cs) &
    (write(abs(cs)) IFF write(cs)) & (ack(abs(cs)) IFF ack(cs)) &
       (next_write(abs(cs)) IFF next_write(cs)) &
          pc(abs(cs)) IFF (pc(cs) = wd)
```

We have also used PVS to establish that this abstraction mapping satisfies the conditions on the abstraction mapping that are actually stronger than those discussed in Section 3. These are shown above for this example. The lemma homo_morphic shows that abstraction abs preserves the nexs-state relation and congruent shows that the atomic predicates used in the property proved are congruent with respect to the equivalence classes introduced by the abstraction on the concrete machine states. There is still a gap in the proof since we have not proved that these abstraction conditions do guarantee property preservation. This fact can also be proved using PVS for abstraction mappings.

5 Conclusions and Future Work

Model checking and theorem proving are complementary verification technologies. Model checking is effective for control-dominated systems with small state spaces, where neither the invariant nor the proof is easily constructed. Theorem proving on the other hand is suitable for data-dominated verification where the state spaces can be large or unbounded.

The combination of these technologies can be exploited in a number of ways. We have illustrated one application where model checking is applied to a finite state abstraction of a system where the abstraction is justified by means of theorem proving. We have studied the example of the asynchronous interaction between a pipelined processor and memory. The main safety property of this system is rather more easily proved by the PVS theorem prover than by model checking. PVS is used to construct a finite state abstraction of the processor-memory system. Model checking applied to this abstraction easily yields that each subsequent opcode is eventually loaded. This example is usually done *informally.*

The combination of theorem proving and model checking has several other uses we are currently exploring. For example, theorem proving can be used to prove general temporal properties that can be combined with temporal properties of specific next-state relations proved using model checking. Theorem proving can be used to prove global system properties by composing local system properties (with smaller state spaces) that have been proved using model checking. Model checking can also be used in the induction step for showing that a property holds of an N-process system. Consider for example, an N-process mutual-exclusion algorithm obtained by recursively selecting a "winner" from $N-1$ processes and using Peterson's algorithm to arbitrate between this winner and the Nth process. By the induction hypothesis, there is at most one winner from the first $N-1$ processes. Ths Nth process does not interfere with this selection. The correctness of the algorithm then follows from the correctness of the 2-process Peterson algorithm which has been verified by model checking.

We argue that the mu-calculus serves as a good basis for combining model checking with theorem proving. The mu-calculus can be used to conveniently define past, future LTL operators, and CTL with fairness constraints. It can also be cleanly defined in PVS so that model checking can be used as a decision procedure for a well-defined fragment of PVS. The mu-calculus has one drawback: the complexity of model checking is exponential in the number of alternations of fixpoint operators, but this is rarely a problem in practice.

Our framework can also be used to integrate PVS with a CTL model checker such as SMV by defining CTL operators in PVS and using SMV as a decision procedure for the CTL fragment. For our present purpose, we have chosen the mu-calculus for its greater expressiveness and for ease of translation. Note that, the model checkable fragment of PVS already provides a richer language than SMV. One disadvantage with respect to SMV is that we are unable, at present, to generate counterexample traces when a property does not hold in a model. This is an important topic for future work. We also plan to investigate the use of the combined technology to explore LTL model checking and verification based on language-containment

References

[1] Julian Bradfield and Colin Stirling. Verifying temporal properties of processes. In J. C. M. Baeten and J. W. Klop, editors, *CONCUR '90*, number 458 in Lecture Notes in Computer Science, pages 115–125. Springer Verlag, 1990.

[2] J. R. Burch, E. M. Clarke, D. E. Long, K. L. McMillan, and D. L. Dill. Symbolic model checking for sequential circuit verification. *IEEE Transactions on Computer-Aided Design*, 13(4):401–424, April 1994.

[3] 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.

[4] E. Clarke, O. Grumberg, and K. Hamaguchi. Another look at LTL model checking. In David Dill, editor, *Computer-Aided Verification 94*, volume 818 of *Lecture Notes in Computer Science*, pages 415–427, Stanford, CA, June 1994. Springer Verlag.

[5] 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.

[6] R. Cleaveland. Tableau-based model checking in the propositional mu-calculus. Technical Report 2/89, University of Sussex, March 1989.

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

[8] 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.

[9] E. Allen Emerson. Temporal and modal logic. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, chapter 16, pages 995–1072. Elsevier and MIT press, Amsterdam, The Netherlands, and Cambridge, MA, 1990.

[10] E.A. Emerson and C.L Lei. Efficient model checking in fragments of the propositional mu-calculus. In *Proceedings of the 10th Symposium on Principles of Programming Languages*, pages 84–96, New Orleans, LA, January 1985. Association for Computing Machinery.

[11] Urban Engberg, Peter Grønning, and Leslie Lamport. Mechanical verification of concurrent systems with TLA. In G. v. Bochmann and D. K. Probst, editors, *Computer-Aided Verification 92*, number 663 in Lecture Notes in Computer Science, pages 44–55. Springer Verlag, 1992.

[12] 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.

[13] Susanne Graf. Verification of a distributed cache memory by using abstractions. In David L. Dill, editor, *Computer-Aided Verification 94*, number 818 in Lecture Notes in Computer Science, pages 207–219. Springer Verlag, 1994.

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

[15] 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.

[16] D. Kozen. Results on the propositional mu-calculus. *Theoretical Computer Science*, pages 333–354, December 1983.

[17] R. Kurshan and L. Lamport. Verification of a multiplier: 64 bits and beyond. In Costas Courcoubetis, editor, *Computer-Aided Veri-*

fication93, volume 697 of *Lecture Notes in Computer Science*, pages 166–179, Elounda, Greece, June/July 1993. Springer Verlag.

[18] R.P. Kurshan. *Automata-Theoretic Verification of Coordinating Processes*. Princeton University Press, Princeton, NJ, 1993.

[19] R.P. Kurshan and K. McMillan. A structural induction theorem for processes. In *8th ACM Symposium on Principles of Distributed Computing*, pages 239–248, Edmonton, Alberta, Canada, August 1989.

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

[21] Steven P. Miller and Mandayam Srivas. Formal verification of the AAMP5 microprocessor: A case study in the industrial use of formal methods. In *WIFT '95: Workshop on Industrial-Strength Formal Specification Techniques*, pages 2–16, Boca Raton, FL, 1995. IEEE Computer Society.

[22] Olaf Müller and Tobias Nipkow. Combining model checking and deduction for I/O automata. Draft manuscript, 1995.

[23] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, June 1992. Springer-Verlag.

[24] D. Park. Finiteness is mu-effable. Technical Report 3, The University of Warwick, March 1989. Theory of Computation Report.

[25] P. Wolper and V. Lovinfosse. Verifying properties of large sets of processes with network invariants. In J. Sifakis, editor, *International Workshop on Automatic Verification Methods for Finite State Systems*, volume 407 of *Lecture Notes in Computer Science*, pages 68–80, Grenoble, France, June 1989. Springer-Verlag.

```
mucalculus[t:TYPE]: THEORY
BEGIN

  IMPORTING orders[pred[t]]
  <=(p1,p2) = FORALL s: p1(s) IMPLIES p2(s)

  ismono(pp) = (FORALL p1,p2: p1 <= p2 IMPLIES pp(p1) <= pp(p2))
  isfix (pp,p) = (pp(p) = p)
  glb(set_of_pred)(s) = (FORALL p: member(p,set_of_pred) IMPLIES p(s))

% least fixpoint
  mu (pp) = glb (LAMBDA p: pp(p) <= p)
  islfp (pp,p1) = isfix (pp,p1) AND
                  FORALL p2: isfix (pp,p2) IMPLIES p1 <= p2
  lub (setofpred)(s):bool = EXISTS p: member(p,setofpred) AND p(s)

% greatest fixpoint
  nu (pp) = lub (LAMBDA p: pp(p) <= p)

  ff: VAR [pred[t]->pred[t]]
  lfp_is_lfp: THEOREM
      ismono(ff) IMPLIES islfp(ff,lfp(ff))
END mucalculus

ctlops[t:TYPE]: THEORY
BEGIN
  IMPORTING mucalculus

  u,v,w: VAR t
  f,g,Q,P,p1,p2: VAR pred[t]

% Next state relation
  N: VAR [t, t->bool]

  EX(N,f)(u):bool =  (EXISTS v: (f(v) AND N(u, v)))
  EG(N,f):pred[t] = nu (LAMBDA Q: (f AND EX(N,Q)))
  EU(N,f,g):pred[t] = mu (LAMBDA Q: (g OR (f AND EX(N,Q))))
  EF(N,f):pred[t] = EU(N,(LAMBDA u: TRUE),f)
  AX(N,f):pred[t] = NOT EX(N, NOT f)
  AF(N,f):pred[t] = NOT EG(N, NOT f)
  AG(N,f):pred[t] = NOT EF(N, NOT f)
  AU(N,f,g):pred[t] = NOT EU(N, NOT g, (NOT f AND NOT g))
                      AND AF(N,g)
  fairEG(N,f)(Ff):pred[t] =
  nu(LAMBDA Q: f AND mu(LAMBDA P: EX(N, f AND ((Ff AND Q) OR P))))
END ctlops
```

Figure 1: Parts of mu-calculus and fairCTL theories in PVS

```
concrete_pipe[addr: TYPE, word: TYPE, opcodes: TYPE]: THEORY

  BEGIN
   ASSUMING
    addr_nonempty: ASSUMPTION (EXISTS (a: addr): TRUE)
    word_nonempty: ASSUMPTION (EXISTS (d: word): TRUE)
    opcodes_nonempty: ASSUMPTION (EXISTS (o: opcodes): TRUE)
   ENDASSUMING

   cstaterec: TYPE =
     [# %% Input wires: acknowledgement signal from memory
        ack: bool,
        %% signals to manage memory transactions
        write: bool,
        maddr, rdword: word,

        %% Internal Registers
          % The Pipeline registers
           pc, pcd, pcdd: word,
           dstnd, dstndd: addr,
           stalld, stalldd: bool, opcoded: opcodes,

            regfile: [addr -> word],  % Register file
            wbreg: word,              % Write back register
            opreg1, opreg2: word #]   % ALU input registers

      s, s1, s2: VAR cstaterec

      %% Next state relation

      storeop?: [opcodes -> bool]
      next_write(s): bool = storeop?(opcoded(s)) & NOT stalld(s)
      instrn_rdy(s): bool = NOT write(s) & ack(s)
      dhld(s): bool = IF write(s) THEN NOT ack(s)
                      ELSE next_write(s) & NOT ack(s) ENDIF

      Nack(s1, s2): bool = (ack(s1) => NOT ack(s2))

      Nwrite(s1, s2): bool =
        write(s2) = IF dhld(s1) THEN write(s1)
                    ELSE next_write(s1) ENDIF

      inc: [word -> word]
      inc_ax: AXIOM FORALL (wd: word): (inc(wd) /= wd)
      Npc(s1, s2): bool =
        pc(s2) = IF dhld(s1) THEN pc(s1)
                 ELSIF instrn_rdy(s1) THEN inc(pc(s1)) ELSE pc(s1) ENDIF

      stall(s): bool = (write(s) OR NOT instrn_rdy(s))
         .....

     N(s1, s2): bool =
       Nack(s1, s2) & Nwrite(s1, s2) & Npc(s1, s2) & ...

  END concrete_pipe
```

Figure 2: Microprocessor Specification

```
abs_pipe: THEORY

  BEGIN
  IMPORTING ctlops, signal

  staterec: TYPE =
            [# write, next_write,
               ack, pc: bool
            #]


  s, s1, s2: VAR staterec

  % memory read/write signal

  instrn_rdy(s): bool = NOT write(s) & ack(s)

  dhld(s): bool = IF write(s) THEN NOT ack(s)
                  ELSE next_write(s) & NOT ack(s) ENDIF

  %% Define next state relation
  Nack(s1, s2):bool = ack(s1) IMPLIES NOT ack(s2)

  Nnext_write(s1, s2): bool =
    (dhld(s1) => next_write(s2) = next_write(s1)) &
    ((NOT dhld(s1) & NOT instrn_rdy(s1)) => NOT next_write(s2))

  Nwrite(s1, s2): bool =
    write(s2) = IF dhld(s1) THEN write(s1)
               ELSE next_write(s1) ENDIF

  NPC(s1,s2):bool =
    (dhld(s1) OR (NOT instrn_rdy(s1)) = (pc(s2) =  pc(s1)))

  N(s1,s2):bool =
     Nack(s1,s2) AND NPC(s1,s2) & Nnext_write(s1, s2) &
       Nwrite(s1, s2)

  ackf: pred[staterec] = (LAMBDA s: ack(s))


  instrn_rdy_liveness: LEMMA
     NOT instrn_rdy(s) & pc(s) IMPLIES
        fairAF(N, LAMBDA (s1): pc(s1) & instrn_rdy(s1))(ackf)(s)

  dhld_liveness: LEMMA
     dhld(s) & pc(s) IMPLIES
        fairAF(N, LAMBDA s1: NOT dhld(s1)
             & pc(s1) )(ackf)(s)


  END abs_pipe
```

Figure 3: Abstract Model of the Microprocessor

```
abs(init_pc_val: word)(cs): staterec =
  (# write := write(cs),
     next_write := next_write(cs),
     ack := ack(cs),
     pc := (pc(cs) = init_pc_val) #)

% abstract model is a homomorphic image of the concrete model
homo_morphic1: LEMMA
  concrete_pipe.N(cs1, cs2)) IMPLIES
     (abs_pipe.N(abs(pc(s1))(cs1), abs(pc(s1))(cs2))

homo_morphic2: LEMMA
     (abs_pipe.N(s1, s2)) IFF EXISTS (cs1, cs2):
          abs(pc(cs1))(cs1) = s1 & abs(pc(cs1))(cs2)= s2 &
                                   concrete_pipe.N(cs1, cs2)

congruent: LEMMA FORALL (wd: word): LET abs = abs(wd) IN
  instrn_rdy(abs(cs)) IFF instrn_rdy(cs) &
    (write(abs(cs)) IFF write(cs)) & (ack(abs(cs)) IFF ack(cs)) &
       (next_write(abs(cs)) IFF next_write(cs)) &
          pc(abs(cs)) IFF (pc(cs) = wd)
```

Figure 4: Homomorphism Correctness Conditions