An Integration of Model-Checking with Automated Proof Checking*

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) 859-6136 Fax: +1 (415) 859-2844

March 9, 1995

Abstract

Automated proof checking tools have been successfully employed for the verification of digital systems. These tools have the advantage of being completely general but suffer from the inherent limits to the efficient automation of expressive logics. If the expressiveness is constrained, it is possible to define certain useful fragments of logic for which efficient decision procedures can be found. The paradigm of model-checking yields an important class of decision procedures for establishing temporal properties of finite automata. 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 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.

*Supported by ARPA under contract PR8556, by NSF Grant CCR-930044, and by NASA under contract NAS1-20334. This paper has been submitted for publication

1 Introduction

Automated theorem provers have been employed successfully in the verification of digital hardware and software systems. In this approach, 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 is very general and applies to a large variety of systems and properties, but there are inherent limits to the efficiency with which expressive general-purpose logics can be fully mechanized. There have been 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 [4], 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 Diagram (BDD) and symbolic model-checking. 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.

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 [14] is an early attempt in this direction. In this combination, the HOL proof checker [11] 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. In [17], the two systems used to verify the multiplier were used independently and not actually integrated.

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 [27] 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. Gupta and Fisher [12] exploit the regularity of an iterated automaton to express them by means of inductive boolean functions that can be verified using BDDs.

The goal of the present work is to present a smooth and tight integration of model-checking with proof checking in the context of the PVS proof checker. 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 deep connection 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 a 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 [21] and language containment systems such as COSPAN.

The rest of the paper is organized as follows: PVS proof checker is described in Section 2. Section 3 describes integration of the model-checker in PVS. In Section 3.1 propositional mu-calculus is introduced. The definition of CTL and fairCTL operators in terms of mu-calculus and their embedding in PVS is discussed in Section 3.2. The translation of a hereditarily finite fragment of the mu-calculus into input acceptable by a model-checker for propositional mu-calculus is shown in Section 3.3. We then present in Section 4 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 verification problem that we encountered during the verification of a commercial processor AAMP5 [22] using PVS. The verification of AAMP5, which contains almost half a million transistors, is too large to be verified entirely by a model-checker. We were able to construct the proofs of correctness of AAMP5 automatically in PVS except for the proofs of a set of control-oriented temporal properties that were too tedious for a theorem prover, but automatic for a model-checker. We argue through our example, that an integrated system would enhance the effectiveness of automatic verification of complex systems. Conclusions are summarized in Section 5.

2 An Overview of PVS

PVS is an environment for writing specifications and developing proofs [25]. It is an effective tool for hardware verification [8] and also serves as a prototype for exploring new approaches to mechanized formal methods. The primary goal of PVS is to combine an expressive specification language with a productive, interactive proof checker that has a reasonable amount of theorem proving power. The PVS specification language builds on a classical typed higher-order logic enhanced with a rich type system. PVS differs from most interactive proof checkers in providing primitive inference rules that are quite powerful. Some of the most commonly used primitive inferences in PVS are implemented using a combination of an efficient rewriter and decision procedures on ground terms. The decision procedures are complete for linear arithmetic inequalities over reals and boolean combinations of equality over uninterpreted functions. PVS also provides the ability to invoke an external BDD-based propositional simplifier as a primitive inference step in a proof.

3 Model-Checking within Theorem Proving

In our integration scheme, CTL operators that are parameterized with respect to a binary next-state relation N are defined in PVS in terms of mu-calculus, which is itself embedded in the logic of PVS. Temporal logic formulas involving a specific interpretation for N, also defined in PVS, are transformed into mu-calculus formulas involving greatest and least fixpoints [2, 9] by rewriting in PVS. For a class of temporal formulas, where N is known to be finite, the simplified mu-calculus formulas are translated and sent to an external BDD-based mu-calculus simplifier to compute greatest and least fixpoints. If the simplifier returns true then the original temporal formula is considered to be proved. The rewriting step, the translation process and the invocation of the simplifier is encapsulated as an automatic proof strategy that can be invoked as a primitive PVS command called `model-check`.

3.1 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[1]. In addition to atomic propositional terms and predicates on propositional terms, propositional terms may contain predicate variables bound by greatest (ν) and least (μ) fixpoint operators. 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 [2,6,10,15, 26]. We closely follow the formal definition of the syntax of propositional mu-calculus from Clarke and others [2] that forms the basis of the model-checker [13] 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 conventional propositional expression or a relational term. A relational term is one of the following:

- Z, a 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 a predicate variable in Σ and P is a relational term formally monotonic in Z (i.e., Z occurs under an even number of negations in P). Similarly, $\nu Z.P$ is the greatest fixpoint of P, and is equivalent to the negation of the least fixpoint of $\neg P[\neg Z]$.

Temporal logics have proved to be useful in specification and verification of properties of concurrent programs and state-machines models [9, 21]. Temporal logics such as CTL with extensions of fairness (fairCTL) and other temporal modalities, and PLTL can be succinctly expressed using the mu-calculus [2, 10] defined above. Additionally, it has been shown that LTL model-checking can be reduced to fairCTL model-checking [3].

3.2 mu-Calculus and fairCTL in PVS

We develop a fixpoint theory based on an ordering by logical implication as follows. The membership of an element in a set is denoted by its characteristic predicate. An ordering on the predicates corresponding to set inclusion is introduced by logical

[1] Quantification on propositional variables serves to succinctly express conjunction and disjunction, and does not enhance expressive power.

implication on the predicates, i.e., $p_1 \leq p_2 \equiv \forall$ s: $p_1(s) \Rightarrow p_2(s)$. Monotonicity is defined as a relation that preserves such an ordering. A fixpoint of a functional[2] `pp`, when applied to a predicate argument equals the predicate itself. The Greatest Lower Bound `glb` of a set `set_of_pred` is the intersection of all members of the set. The Least Upper Bound `lub` of a set `set_of_pred` is the union of all members of the set. The least fixpoint (`mu`) of a monotonic functional `pp` is the `glb` of the set of functions satisfying the property that the application of the functional to a predicate `p` is less than or equal to the predicate `p` itself. The greatest fixpoint (`nu`) of a monotonic functional `pp` is the LUB of the set of functions satisfying the property that the application of the functional to a predicate `p` is greater than `p` itself:

```
mu (pp) = glb (LAMBDA p: pp(p) <= p)
nu (pp) = lub (LAMBDA p: pp(p) <= p)
```

The property that predicates ordered by logical implication constitutes a complete lattice has been shown in PVS.

The mu-calculus theory thus developed is used to develop the PVS theory (shown in Table 2 of the Appendix) for fairCTL. The fixpoint and fairCTL theories are parametrized by an uninterpreted type `t`. In a state-machine model specification, the type `t` is instantiated to the type of the state. In fairCTL theory, first the next state relation `N` is defined as a binary predicate on a pair of states. The temporal operators `EX`, `EG`, `EU`, and `EF` with existential path quantifiers are defined in a standard manner, and the corresponding operators `AX`, `AF`, `AG` and `AU` with universal path quantifiers are defined by introducing appropriate negations in the corresponding existential versions. An example definition for `EG` is the following:

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

3.2.1 Fairness

A fairness constraint expresses a condition that should hold sufficiently often along a path [5]. Fairness constraints cannot be directly expressed in CTL. We use the more expressive mu-calculus to specify the fair temporal logic operators as follows. We first define `fairEG(N,f)(Ff)`:

```
fairEG(N,f)(Ff):pred[t] =
   nu(LAMBDA Q: f AND mu(LAMBDA P: EX(N, f AND ((Ff AND Q) OR P))))
```

[2] A *functional* is a function that may take functions as arguments or return functions as values.

"there exists a computation path in the state machine model defined by the next state relation `N`, in which a fair formula `Ff` holds infinitely often and formula `f` holds globally along the path", where `N` is the next state relation, `Ff` is the formula that has to hold infinitely often, and `f` is the formula that has to hold globally on all such fair paths. The definition of fairness we have used above is based on the work by Emerson and Lei [10], which also allows other notions of fairness[3]. The other fairCTL operators are defined by using the `fairEG` operator in the same manner as Burch et al. [1, 21]. 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 might lead to empty models in which every property holds trivially.

3.3 Translation from PVS to mu-calculus

The PVS specification language [24] features basic types such as reals, integers, naturals, and lists, and a rich set of common programming language constructs such as scalars, arrays, functions, and records for structuring specifications. However, since the low-level BDD-based mu-calculus model-checker accepts only the language of propositional mu-calculus as discussed in Section 3.1, an automatic translation is provided from a fragment of the PVS language to propositional mu-calculus.

The fragment of the PVS language that is translated into mu-calculus consists of expressions whose types are hereditarily finite: i.e., types which are either themselves finite or constructed from expressions which are of finite types. However, boolean expressions in which subexpression types are not heriditarily finite are translated into a single boolean variable. For example, the equality `x = 1`, where `x` is a variable of type `natural`, is translated into a simple boolean variable `bvar_x1`. This scheme ensures that the output of translation is always a legal mu-calculus expression even when the input contains a subterm of non-finite type.

Let the syntactic category `t-expr` denote the set consisting of *propostional expressions*, *scalars*, *records*, *tuples*, *arrays* of a specified finite length, *equalities* on `t-expr`s, *conditional expressions* on `t-expr`s, `t-expr`s with quantifications on boolean variables, and `t-expr`s quantified by μ or ν. In the following, we give an abstract description of the translation in terms of the function `pvs-to-mu`. The input to `pvs-to-mu` is either a `t-expr` belonging to the PVS language, or a list whose elements are either `t-expr`s or nested lists of `t-expr`s. The output of `pvs-to-mu` is a mu-calculus formula defined in Section 3.1.

[3] The mu-calculus expression for `fairEG` by Burch et al. [1, 21] allows an unfair path.

1. `pvs-to-mu(prop-expr) = prop-expr`, where `prop-expr` is a propositional expression. Thus, `true`, `false`, boolean variables, and boolean expressions consisting of only simple boolean variables are translated without change.

2. `pvs-to-mu(scalar) = binary-encoding(scalar)`, where a `scalar` is either a variable or a constant of enumerated type, and `binary-encoding` is the corresponding boolean representation of the scalar in a binary encoding scheme. For example, let `instr: TYPE = {jmp,mov,add}` be a scalar type, and `x` be a scalar variable of `instr` type. Scalar constants `jmp`, `mov`, and `add` are translated as lists `[true,true]`, `[true,false]`, and `[false,false]`, respectively. The scalar variable `x` is translated as a list `[bvar1,bvar2]`.

3. `pvs-to-mu(nested_list_of_t-expr) = flattened_tuple_of_t-expr`
   For example, `[[bvar1,bvar2],bvar3]` is translated as `bvar1,bvar2,bvar3`. Such a translated tuple will eventually be incorporate as part of arguments to predicates, and as variables bound by λ and quantifiers. Thus `P(x,x)` is translated as `P(bvar1,bvar2,bvar1,bvar2)`.

4. `pvs-to-mu(record) = [ pvs-to-mu(field1),...,pvs-to-mu(fieldN)]`, where
   `record = [# field1,...,fieldN #]` and `field1,...,fieldN` are `t-exprs`. A `tuple` is translated in a similar manner. For example, let `state: TYPE = [# x1,x2: instr #]` be a record type, and `s1: VAR state` be a record variable. The translation output gives a list `[bvar-x1,bvar-x1,bvar-x2,bvar-x2]`. This will eventually be translated into a mu-calculus expression via rules 7,9, and 10.

5. `pvs-to-mu(array([0...N]`$\rightarrow$`fType))=` `[ pvs-to-mu(el1),...,pvs-to-mu(elN)]` where the range type `fType` is hereditarily finite, `el1,...,elN` are the elements of the array, and `N` is specified and finite.

6. `pvs-to-mu(t-expr1 = t-expr2) = (pvs-to-mu(t-expr1) = pvs-to-mu(t-expr2))`.

7. `pvs-to-mu(nested_list_of_t-expr_1 = nested_list_of_t-expr_2) =` $\bigwedge_i$ `pvs-to-mu(t-expr1[i] = t-expr2[i])`, i.e., equality on lists is translated to the conjunction of the equalities on the elements of the flattened lists.

8.
```
pvs-to-mu(t-exprs = IF test THEN then-t-exprs ELSE else-t-exprs) =
        IF test THEN pvs-to-mu(t-expr = then-t-exprs)
                ELSE pvs-to-mu(t-expr = else-t-exprs).
```

i.e., the inner `IF` expressions are lifted to the top.

9. `pvs-to-mu(`$\forall x$`.t-expr) =` $\forall$ `pvs-to-mu(x). pvs-to-mu(t-expr).` This translation also holds for $\exists$ and λ binders in place of $\forall$.

10. `pvs-to-mu(`$\mu.P(e_1, e_2, \ldots, e_n)$`)` = $\mu.P($`pvs-to-mu(`e_1`)`,`pvs-to-mu(`e_2`)`$, \ldots,$ `pvs-to-mu(`e_n`)`$)$, where $e_1, e_2, \ldots, e_n$ are `t-exprs`. This translation also holds for ν binder in place of μ.

Other constructions such as *functions* and *subtypes* of hereditarily finite types could also be translated, but are not yet implemented in our translation scheme. 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 [21], the PVS language is more expressive, and has the significant advantage of a proof system for the language.

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 [22].

We verify a property of a small microprocessor CPU design that is an extension of the CPU example used by Burch, et al., [2] to illustrate the power of symbolic model checking. The example in [2] is a register-transfer level design of the datapath and controller of a microprocessor that executes instructions of the form `(opcode src1 src2 dstn)`, i.e., "destination register `dstn` in the register file `REGFILE` becomes some `ALU` function determined by `opcode` of the contents of source registers `src1` and `src2`. The processor uses a three-stage pipeline to simultaneously execute three distinct stages of three successive instructions: *read*, *compute*, and *write*. The example assumed that the various fields of an instruction were fetched from the memory and presented as inputs to the design. The main objective of the exercise in [2] was to ensure that the pipeline control logic worked correctly by verifying the following property shown as formulated in [2]. In the following, AX^n `regfile` is used as an abbreviation to denote the state of `regfile` n cycles from the current state.

```
NOT stall IMPLIES (AX^3 regfile)(dstn) =
                    aluop(opcode, (AX^2 regfile)(src1), (AX^2 regfile)(src2))
```

The property formalized above states that an instruction entering the pipeline in the current state gets completed correctly, i.e., will write the correct result into the register file `regfile`, three transitions (cycles) later, provided the instruction is not aborted (`NOT stall`). This property was verified in [2] using symbolic model checking for different word sizes and different number of registers in the register file. In [8, 23], we showed that the same property in its general form can be verified automatically in PVS. The verification in PVS employs a simple strategy that combines PVS's automatic rewriting, BDD-based propositional simplification and arithmetic decision procedures. We have used this strategy successfully to automate verification of several other microprocessor and hardware design correctness properties [8, 22]. A theorem prover is better-suited and significantly more efficient than a model-checker for verifying such a property because it is possible to prove the property for arbitrary word size and registers in a theorem prover.

On the other hand constructing proofs of temporal properties of interacting state machines is harder using a theorem prover, especially when liveness issues are involved. Verification of realistic microprocessor designs, in which the CPU implements its memory operations by means of an interactive hand-shake protocol with the external memory, involves verifying both temporal properties and data transformation properties. Although the number of states in such a design is too large for a model-checker, the temporal properties needed in the verification can often be proved using an abstraction of the design that is finite state, and hence, can be handled by model-checker. An integrated theorem proving-model-checker setup can not only help formalize the refinement process needed for such a verification approach but also automate the verification of the temporal properties.

To illustrate the combined approach to verification, we extend the CPU design in [2] by adding appropriate logic to implement a `store` instruction and instruction fetching. A block diagram of the CPU is shown in Figure 1. The `store` instruction stores the contents of the register indexed by `src1` at the address given by the contents of the register indexed by `src2` ignoring the `dstn` field. The extra logic in the design sends out read (for instruction fetches) and write (for store instruction) requests to an external memory. 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.

In the PVS specification (shown in Figure 3 of the Appendix) of the CPU, the state of the CPU is modeled as a record type, which has components to manage

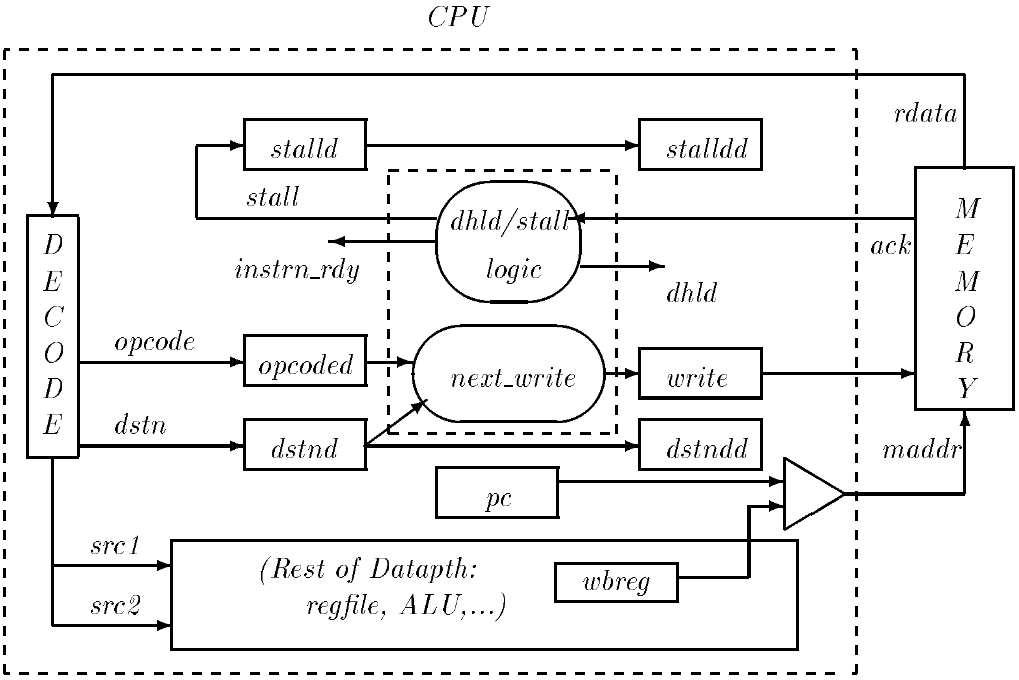

Figure 1: The Microprocessor System

transactions with the memory and internal registers to manage the pipeline stages. The acknowledgment signal (`ack`) from the memory is also included as a component of the state, but constrained minimally. It 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 signal `write` being true indicates a pending write request to memory. The signal is issued by the instruction in the write stage of the pipeline when the instruction is of store class and not meant to be stalled. It is computed in advance as the signal `next_write` to prevent a possible clash with a pending read or write. A read request for instruction fetching is assumed to be active whenever `write` is false.

The stalling of the pipeline when the CPU is waiting for an acknowledgment from the memory is implemented as follows. If the CPU is waiting for a data write to complete the data path state is *held* (indicated by `dhld`), i.e., the state is frozen, until `ack` is true. If the CPU is waiting for a read to complete in response to an instruction fetch then the CPU is *stalled*, i.e., it streams in a series of NOP instructions by making the `stall` signal true until an acknowledgment is received.

Note that the duration for which the CPU is held or stalled is indefinite since we have left the response time of the memory arbitrary, although finite. Hence the correctness property of the CPU must be expressed as an eventuality property. For example, the property that the opcode of the instruction following the current one is guaranteed to move into the appropriate pipeline registers `opcoded` is expressed in PVS as follows.

```
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)
```

Note that the property is expressed using `fairAF` operator with the predicate `ack` as a fairness predicate, since the eventuality property holds only for fair paths in which the acknowledgment becomes true infinitely often. A proof of the above property involves reasoning about the behavior of the CPU when it is either held or stalled. Specifically, we need the following properties.

```
      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)
```

`Fetch_completes` asserts that the CPU will not stall for ever and that the program counter `pc` remains unchanged when stalling. `Instrn_rdy` is a bit maintained in the design to remember whether the instruction register contains a valid next instruction. `Write_completes` asserts a similar property about the behavior when the CPU is held.

While it is conceptually easy, at least in the current example, to see why `fetch_completes` is true, constructing a proof of it in a theorem prover can be quite tedious. If a new instruction is not ready, it is either because the memory has not responded to a read request for an instruction fetch or because the CPU is waiting for data write in progress to complete to send a read request. In the first case, the liveness of memory guarantees termination of the loop. In the second case, the CPU cannot send an infinite sequence of write requests because when an instruction is not ready the CPU keeps sending a NOP through the pipeline which does not generate a write request. A proof has to use induction on the length of the time it takes for memory to respond with an acknowledgment after a memory request. Complicating the induction proof is the fact that the stalling loop and the data holding loop are interdependent. So the proof has to use simultaneous induction on two loops.

An alternative approach is to use model-checking to prove the `fetch_completes` and `write_completes` lemmas. These two properties in their present form cannot be proved using model-checking because the property contains components of the state, eg. `pc`, that are either of an uninterpreted type or dependent on another component that is of an uninterpreted type, i.e. not hereditarily finite. We show how they can be proved by abstracting the design to a finite state machine in Section 4.2. Once we have the `fetch_completes` property as a lemma `next_instrn_entry` is proved in PVS by rewriting using the next state relation and the memory axioms. Note that a property of the form `AX` that relates states separated by a finite distance can be proved automatically using rewriting in PVS.

4.1 Proving the Lemmas Using Model-Checking

To prove the lemmas using the model checking connection, it is necessary to execute the following steps.

1. Define an abstract finite state model of the design (the *concrete model*) that "preserves" the validity. That is, the property of interest must be valid in the abstract model if it is also valid in the concrete model.

2. Prove that the abstract finite model preserves the validity of the concrete model. For this, as has been shown[4] by E.M.Clarke, et al, in [4], the abstract model must be an *homomorphic image* of the concrete model for the property being verified. For a specific example, this proof can be done as follows.

   Homomorphism Property Define an abstraction function (`ABS`) from the state space of the concrete design to the state space of the abstract model so that the following conditions hold. In the following proof goal, `N_A` and `N_C` denote the next-state relations of the abstract and concrete models, respectively.

   ```
   FORALL (a1, a2: A): N_A(a1, a2) IFF
      (EXISTS (c1, c2 : C): a1 = ABS(c1) AND a2 = ABS(c2)
                                         AND N_C(c1, c2))).
   ```

[4]The proof of the abstraction result in [4] can be mechanized in PVS using the embedding of mu-calculus, although we have not done it yet.

Congruence Property For every primitive predicate `p_c` appearing in the temporal (CTL) formula being proved define a corresponding abstract predicate `p_a` on the abstract state space so that the following condition holds:

```
FORALL (c: C): p_a(ABS(c)) IFF p_c(c)
```

3. Prove `p_a` by invoking the model-check command.

Note that once the user constructs a definition of `ABS` and an abstract model `N_A`, the requirements on the abstraction function shown in step 2 can be generated automatically. Since PVS supports a higher-order logic and a language for building strategies, it is possible to package the refinement process described above into a strategy that takes `ABS` and `N_A` as parameters. The strategy can be designed so that invoking it on a succedent formula in a proof goal would create the requirements to be proved on `ABS` and the translated formula `p_a` as subgoals to be discharged. We have not yet implemented such a strategy in PVS. In the following section, we illustrate performing the steps enumerated above by hand on the microprocessor example.

4.2 Abstracting the Microprocessor Design

The properties to be verified using model-checking determine the information to be retained in the abstract model. In the present case, the `fetch_completes` and `write_completes` lemmas express properties about the signals `instrn_rdy`, `dhld`, and `pc`. So, in the abstract model it is necessary to retain only those components of the state of the concrete model that influence these signals. The relevant components are `ack`, `write`, `stalld`, `opcoded`, and `pc`. We combine the required part of the information contributed by `stalld` and `opcoded` (available in the concrete version of `next_write`) into a single component, `next_write`, in the abstract model. 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 abstract model is less deterministic than the concrete model. The updating of `write` and `ack` are identical to their concrete counterparts. The updating of PC captures the conditions under which the program counter is not changed in the concrete model. The translation of the concrete properties to be proved into an abstract form is straightforward in the present case and are shown in the Appendix. Both the translated properties are proved automatically for the abstract next-state relation using the model-check command. The homomorphism and congruence properties are proved automatically in PVS by rewriting the definitions of the abstraction function and the next-state relations. User intervention is needed in the proof of the homomorphism property in one direction to instantiate the existential quantifier with the appropriate concrete state record. A PVS specification of the abstract model, the definition of the abstraction function, and the homomorphism and congruence conditions proved are shown in the Appendix.

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

Acknowledgments

We thank John Rushby for encouraging us to write this paper and providing comments on earlier drafts of the paper. We are grateful to Geert Janssen for supplying us the BDD-based mu-calculus model-checker.

References

[1] 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.

[2] J. R. Burch, E. M. Clarke, K. L McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *5th Annual IEEE Symposium on Logic in Computer Science*, pages 428–439, Philadelphia, PA, June 1990. IEEE Computer Society.

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

[4] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. In *19th ACM Symposium on Principles of Programming Languages*, pages 343–354, Albuquerque, NM, January 1992. Association for Computing Machinery.

[5] E.M. Clarke and Ö. Grumberg, editors. *Annual Review of Computer Science, Volume 2*. Annual Reviews, Inc., Palo Alto, CA, 1987.

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

[7] Costas Courcoubetis, editor. *Computer-Aided Verification, CAV '93*, volume 697 of *Lecture Notes in Computer Science*, Elounda, Greece, June/July 1993. Springer-Verlag.

[8] D. Cyrluk, S. Rajan, N. Shankar, and M. K. Srivas. Effective theorem proving for hardware verification. In Kumar and Kropf [16], pages 287–305.

[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] 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.

[12] A. Gupta and A. Fisher. Parametric circuit representation using inductive boolean functions. In Courcoubetis [7], pages 15–28.

[13] G. L. J. M. Janssen. *ROBDD Software*. Department of Electrical Engineering, Eindhoven University of Technology, October 1993.

[14] 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.

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

[16] Ramayya Kumar and Thomas Kropf, editors. *Preliminary Proceedings of the Second Conference on Theorem Provers in Circuit Design*, Bad Herrenalb (Blackforest), Germany, September 1994. Forschungszentrum Informatik an der Universität Karlsruhe, FZI Publication 4/94.

[17] R. Kurshan and L. Lamport. Verification of a multiplier: 64 bits and beyond. In Courcoubetis [7], pages 166–179.

[18] R. P. Kurshan. Analysis of discrete event coordination. In *Proceedings of the REX Workshop on Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness*, volume 430 of *Lecture Notes in Computer Science*. Springer-Verlag, 1990.

[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] Patrick Lincoln, Sam Owre, John Rushby, N. Shankar, and Friedrich von Henke. Eight papers on formal verification. Technical Report SRI-CSL-93-4, Computer Science Laboratory, SRI International, Menlo Park, CA, May 1993.

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

[22] 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*, Boca Raton, FL, 1995. IEEE Computer Society. To appear.

[23] S. Owre, J. M. Rushby, N. Shankar, and M. K. Srivas. A tutorial on using PVS for hardware verification. In Kumar and Kropf [16], pages 167–188.

[24] S. Owre, N. Shankar, and J. M. Rushby. *The PVS Specification Language (Beta Release)*. Computer Science Laboratory, SRI International, Menlo Park, CA, February 1993.

[25] S. Owre, N. Shankar, and J. M. Rushby. *User Guide for the PVS Specification and Verification System (Beta Release)*. Computer Science Laboratory, SRI International, Menlo Park, CA, February 1993. Three volumes: Language, System, and Prover Reference Manuals.

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

[27] 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 2: 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 3: 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 4: 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 5: Homomorphism Correctness Conditions