Automatic Generation of Invariants in Processor Verification

Jeffrey X. Su, David L. Dill, and Clark W. Barrett

Computer Science Department
Stanford University

Abstract. A central task in formal verification is the definition of *invariants*, which characterize the reachable states of the system. When a system is finite-state, invariants can be discovered automatically.
Our experience in verifying microprocessors using symbolic logic is that finding adequate invariants is extremely time-consuming. We present three techniques for automating the discovery of some of these invariants. All of them are essentially syntactic transformations on a logical formula derived from the state transition function. The goal is to eliminate quantifiers and extract small clauses implied by the larger formula.
We have implemented the method and exercised it on a description of the FLASH Protocol Processor (PP), a microprocessor designed at Stanford for handling communications in a multiprocessor. We had previously verified the PP by manually deriving invariants.
Although the method is simple, it discovered 6 out of 7 of the invariants needed for verification of the CPU of the processor design, and 28 out of 72 invariants needed for verification of the memory system of the processor. We believe that, in the future, the discovery of invariants can be largely automated by a combination of different methods, including this one.

1 Introduction

As microprocessors become more complex, an increasingly large fraction of the design time is spent on validation. Validation techniques which rely entirely on simulation are limited because of the many possible cases which must be accounted for. Sometimes, significant bugs are found after a processor is commercially released, which is both embarrassing and costly.

Formal verification techniques have been steadily improving over the last decade, and several simple microprocessors have been verified [3, 6, 8]. The methods used in these verification efforts are based on theorem provers, which require a great deal of expert guidance. In addition, some automatic techniques used on simple processors (i.e. [9]) are not applicable to pipelined processors. And even though some pipelined processors have been successfully verified [4, 14, 15, 17], they are either very simple or require a great deal of work to verify.

Burch and Dill [1] proposed a new method for verifying the control of microprocessors. The verification method compares two behavioral descriptions of the processor: a pipelined implementation and a simpler, unpipelined specification. A symbolic simulator is used to generate a collection of formulas in a quantifier-free fragment of first-order logic including propositional connectives, equality, and uninterpreted functions. These formulas are checked for universal truth by a *validity checker*. For an incorrect implementation, this validity checker can produce a specific example where the implementation of the processor contradicts its specification. Significant effort has gone into making this validity checker fast and efficient [7, 18].

For all but the simplest descriptions, the proofs require *invariants*: a logical formula characterizing a superset of the states reachable from the initial state of the processor. Currently, finding appropriate invariants is the single most labor-intensive part of the verification method.

The most complex design to which we have applied the method is the Stanford Protocol Processor (PP) [16], designed by the Stanford FLASH group. The proof was divided into two parts: verification of the CPU and verification of the memory system. Most of the invariants had to be found by trial and error: we attempted verification; when it failed, we analyzed the results to see if the problem was a design error (very rarely), an error from translating the original design to our input language (frequent), an error in the invariant (frequent), or an invariant that was too weak (frequent). This process took a few weeks for the CPU, and a few months for the memory system. It became clear from this experience that even partial automation of invariant discovery would be a major step towards making processor verification more practical.

The invariant discovery methods described here were inspired by inspecting the logical formulas involved in the verification of the PP. The method is to find a formula characterizing the set of states reachable from at least one other state. The disjunction of this formula with a formula characterizing the initial state is an invariant, as is anything implied by the disjunction. However, the formula has existential quantifiers, which is a problem since we are working in a quantifier-free logic. So we employ several simple transformations to extract quantifier-free formulas that are implied by the original formula (and, thus, also invariants).

Perhaps the most surprising result is that, although the method is seemingly weak, it finds a significant number of the invariants that are needed in the proof of the PP. Indeed, it discovers 6 out of the 7 invariants required for the CPU verification, and 28 out of 72 invariants required in the memory system. From this, we can reasonably conclude that the difficulty of verifying the PP would have been greatly reduced if we had used this method to find the invariants instead of doing it manually. However, we believe that the the greatest potential of the method will be realized when it is used in combination with other invariant discovery techniques.

The rest of the paper is organized as follows: related work in invariant generation is discussed in Section 2. The logic and processor model are described in Section 3. Our methods for automatically finding invariants along with proofs of their correctness are presented in Section 4. Experimental results are provided in Section 5, and some concluding remarks are given in Section 6.

2 Generating Invariants

There are existing methods to find invariants automatically. Manna and Pnueli show how to generate invariants for proving safety properties in fair transition systems [11], and these methods have been extended by Bensalem et al [13]. Their methods can be classified as either top-down or bottom-up.

Top-down invariant generation begins with an assertion that we desire to prove for a given system. If this assertion is not valid in general, then various heuristics are applied to strengthen the assertion. However, this method is not guaranteed ever to produce a valid assertion. Bottom-up methods, on the other hand, simply look at the system and use it to deduce assertions which are always valid. Bottom-up invariants tend to be simple properties of the system, and few practically interesting properties are likely to be generated using only bottom-up methods. The two methods can be combined by using invariants generated by the bottom-up method as strengthening assertions for a top-down approach. The Stanford Temporal Prover (STeP) is a system which implements these methods in a tool for program verification [10]. Many properties of simple examples can be proved with very little manual effort by using the automatic methods provided by STeP.

Although there are inherent differences between our verification model and that used by STeP, the ideas of top-down and bottom-up invariant generation are applicable to the approach we take. Ours are bottom-up syntactic methods for extracting simple invariants from the state transition function of an implementation. These invariants are then used to limit the initial state space in our verification model. If the automatic invariants are not sufficient, additional assertions can be added manually in a top-down manner until a valid verification condition is obtained.

3 Logic and Processor Model

A processor is modeled as a Mealy machine M=$(Q, \Sigma, \Delta, \delta, \eta, q_0)$, where Q is a finite set of states, $q_0 \in Q$ is the initial state, Σ and Δ are finite sets of inputs and outputs, δ is the state transition function which maps $Q \times \Sigma$ to Q, and η is the output function which maps $Q \times \Sigma$ to Δ. We require δ to be total (defined for all inputs and states). In the rest of the paper, we will assume that M is given.

The techniques we use to find invariants are based on syntactic manipulation of formulas in first-order logic. We will use the standard Boolean connectives as well as an if-then-else (ite) construct: we write $(ite\ \alpha\ \beta\ \gamma)$ to mean *if* α *then* β *else* γ. Within the formulas, we will use the following variable conventions: $\mathbf{x} \in \Sigma$, and $\mathbf{w}$, $\mathbf{z} \in Q$, where $\mathbf{w}$ is the old or previous state and $\mathbf{z}$ represents the new or current state.

We also assume that δ can be expressed as a formula in quantifier-free first-order logic (we obtain this formula by symbolic simulation). This formula is dependent on a state $\mathbf{w}$ and an input $\mathbf{x}$. We will denote the syntactic representation of the state transition function depending on variables $\mathbf{w}$ and $\mathbf{x}$ by $\tau[\mathbf{w}, \mathbf{x}]$.

We will deal primarily with Boolean logical formulas containing a single free variable which can range over the states of M. Such a formula will be called a *state* *predicate* and will be written as a symbol followed by the state variable (usually $\mathbf{z}$) in square brackets (i.e. $I[\mathbf{z}]$). As a special case, we define the *initial state predicate* as:

$$Q_0[\mathbf{z}] \equiv \mathbf{z} = q_0.$$

We also define the *predecessor state predicate* as:

$$P[\mathbf{z}] \equiv \exists \mathbf{w}, \mathbf{x}.\ \mathbf{z} = \tau[\mathbf{w}, \mathbf{x}].$$

Intuitively, $P[\mathbf{z}]$ holds exactly when **z** has at least one predecessor.
A state predicate which is true for all reachable states is called an *invariant*. Formally,

Definition 1 (Invariant) *Let $I[\mathbf{z}]$ be a state predicate. $I[\mathbf{z}]$ is an invariant if:*
Base Rule*: $Q_0[\mathbf{z}] \Rightarrow I[\mathbf{z}]$, and*
Induction Rule*: $I[\mathbf{w}/\mathbf{z}] \Rightarrow I[\tau[\mathbf{w}, \mathbf{x}]/\mathbf{z}]$* [1]
are valid logical formulas (i.e. true for all possible interpretations of free variables). $\square$

By this definition, the predecessor state predicate is an invariant. In practice, however, it is both difficult and computationally expensive to deal with invariants which contain quantifiers (especially in automatic methods). Thus we seek to find equivalent or weaker invariants which are implied by the predecessor state predicate and which do not contain quantifiers. We use the following theorem as the basis for three techniques to automatically generate such invariants.

Theorem 1. *If $I[\mathbf{z}]$ is a state predicate and $\forall \mathbf{z}.\ P[\mathbf{z}] \Rightarrow I[\mathbf{z}]$ is valid, then $Q_0[\mathbf{z}] \vee I[\mathbf{z}]$ is an invariant.*

Base Rule : Clearly, $Q_0[\mathbf{z}] \Rightarrow Q_0[\mathbf{z}] \vee I[\mathbf{z}]$ is valid.
Induction Rule : We must show that $Q_0[\mathbf{w}/\mathbf{z}] \vee I[\mathbf{w}] \Rightarrow Q_0[\tau[\mathbf{w}, \mathbf{x}]/\mathbf{z}] \vee I[\tau[\mathbf{w}, \mathbf{x}]/\mathbf{z}]$ is valid. Consider just the formula $I[\tau[\mathbf{w}, \mathbf{x}]/\mathbf{z}]$ and let q and σ be arbitrary assignments to $\mathbf{w}$ and $\mathbf{x}$ respectively. We know that $P[\mathbf{z}] \Rightarrow I[\mathbf{z}]$ is valid. By inspection, we see that $P[\mathbf{z}]$ is true under the assignment $\mathbf{z} \leftarrow \delta(q, \sigma)$. It must then be the case that $I[\mathbf{z}]$ is also true under the same assignment. But this is equivalent to $I[\tau[\mathbf{w}, \mathbf{x}]/\mathbf{z}]$ since τ simply computes δ. Thus $I[\tau[\mathbf{w}, \mathbf{x}]/\mathbf{z}]$ is valid and it follows that $Q_0[\mathbf{w}/\mathbf{z}] \vee I[\mathbf{w}] \Rightarrow Q_0[\tau[\mathbf{w}, \mathbf{x}]/\mathbf{z}] \vee I[\tau[\mathbf{w}, \mathbf{x}]/\mathbf{z}]$ is valid.

$\square$

Note that we did not use the hypothesis in the induction step. Thus, as was previously mentioned, these are weak invariants.

[1] We use the notation $I[\mathbf{y}/\mathbf{z}]$ to denote the result of substituting **y** for **z** in $I[\mathbf{z}]$.

4 Automatic Generation of Invariants

In the previous section, for the sake of simplicity, we assumed that the state of a processor can be represented by a single state variable. In practice, it is much more useful to think of the state as being divided into many individual state variables. In the following, we will assume that states are actually n-tuples, so that any state $\mathbf{z}$ can be written as $(z_1, ..., z_n)$ and $\tau[\mathbf{w}, \mathbf{x}]$ can be written as $(\tau_1[\mathbf{w}, \mathbf{x}], ..., \tau_n[\mathbf{w}, \mathbf{x}])$. We will also assume that $\mathbf{w} = (w_1, ..., w_n)$ and $\mathbf{x} = (x_1, ..., x_m)$. We can then write the predecessor state predicate as follows:

$$P[z_1, ..., z_n] = \exists \mathbf{w}, \mathbf{x}.\ \bigwedge_{i=1}^{n} (z_i = \tau_i[\mathbf{w}, \mathbf{x}]).$$

We now use this formula to find valid quantifier-free state predicates. This is done by applying some simple syntactic transformations with the goal of eliminating the existentially quantified state and input variables (which we refer to as bound variables in the following discussion). These methods are discussed in the following subsections.

4.1 Method 1

Our first method is to replace subexpressions containing bound variables in $P(\mathbf{z})$ with state variables wherever possible. We accomplish this by discovering common subexpressions which are equivalent to the state transition function for some state variable. Formally, for each clause $z_i = \tau_i[\mathbf{w}, \mathbf{x}]$ in $P[\mathbf{z}]$, whenever $\tau_i[\mathbf{w}, \mathbf{x}]$ appears as a subexpression of $\tau_j[\mathbf{w}, \mathbf{x}]$ where $i \neq j$, we replace the occurrence of $\tau_i[\mathbf{w}, \mathbf{x}]$ in $\tau_j[\mathbf{w}, \mathbf{x}]$ with z_i. This results in a new set of state transition formulas which depend not only on the input and previous state variables, but on the current state variables as well. We will denote these new formulas by $(\tau'_1[\mathbf{w}, \mathbf{x}, \mathbf{z}], ..., \tau'_n[\mathbf{w}, \mathbf{x}, \mathbf{z}])$. We can then write a new predecessor state predicate $P'[\mathbf{z}]$ which is logically equivalent to $P[\mathbf{z}]$:

$$P'[\mathbf{z}] = \exists \mathbf{w}, \mathbf{x}.\ \bigwedge_{i=1}^{n} (z_i = \tau'_i[\mathbf{w}, \mathbf{x}, \mathbf{z}]).$$

This new predicate $P'[\mathbf{z}]$ has fewer occurrences of bound variables.

Now, if there are any clauses in $P'[\mathbf{z}]$ in which all bound variables have been eliminated as a result of the transformation, they can be moved outside the scope of the existential quantifiers. Call the conjunction of all such clauses $I_1[\mathbf{z}]$ and the remaining term $P_1[\mathbf{z}]$ so that $P'[\mathbf{z}] = I_1[\mathbf{z}] \wedge P_1[\mathbf{z}]$. For simplicity, we assume that the removed clauses are those associated with the highest numbered state variables so that:

$$P_1[\mathbf{z}] = \exists \mathbf{w}, \mathbf{x}.\ \bigwedge_{i=1}^{n_1} (z_i = \tau'_i[\mathbf{w}, \mathbf{x}, \mathbf{z}]).$$

where $n_1 \leq n$.

Since $I_1[\mathbf{z}]$ is a subexpression of $P'[\mathbf{z}]$, it follows that $P'[\mathbf{z}] \Rightarrow I_1[\mathbf{z}]$ is valid, and thus $Q_0[\mathbf{z}] \vee I_1[\mathbf{z}]$ is an invariant by Theorem 1. We will now apply this method to the circuit in Figure 1.

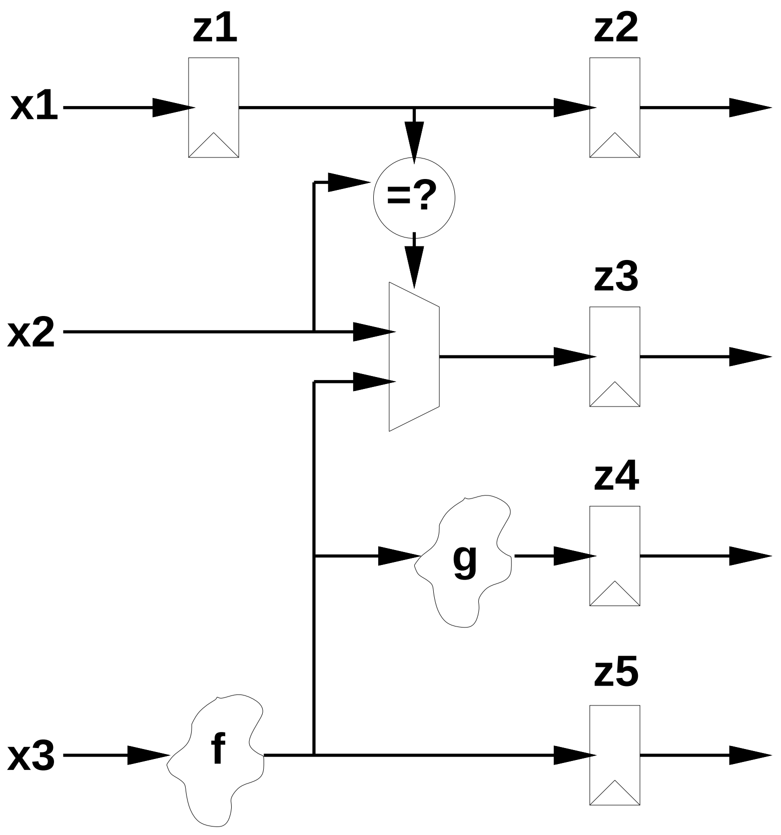

Fig. 1. A simple circuit.

Example 1 *The predecessor state predicate of this circuit is given below (recall that* **w** *represents the previous state and* **z** *represents the current state):*

$$\exists \mathbf{w}, \mathbf{x}.\ [z_1 = x_1\ \wedge\ z_2 = w_1\ \wedge\ z_3 = (ite\ (x_2 = w_1)\ x_2\ f(x_3)) \\ \wedge\ z_4 = g(f(x_3))\ \wedge\ z_5 = f(x_3)]$$

where f and g represent functions computed by combinational logic.
We now substitute state variables z_2 and z_5 for w_1 and $f(x_3)$:

$$\exists \mathbf{w}, \mathbf{x}.\ [z_1 = x_1\ \wedge\ z_2 = w_1\ \wedge\ z_3 = (ite\ (x_2 = z_2)\ x_2\ z_5) \\ \wedge\ z_4 = g(z_5)\ \wedge\ z_5 = f(x_3)]\ .$$

Since the clause $z_4 = g(z_5)$ does not have any bound variables and is the only such clause, we factor it out:

$$\exists \mathbf{w}, \mathbf{x}.\ [z_1 = x_1\ \wedge\ z_2 = w_1\ \wedge\ z_3 = (ite\ (x_2 = z_2)\ x_2\ z_5) \\ \wedge\ z_5 = f(x_3)]\ \wedge\ z_4 = g(z_5).$$

Let $\mathbf{z} = \mathbf{q_0}$ be the initial state predicate for this circuit. Then

$$\mathbf{z} = \mathbf{q_0}\ \vee\ z_4 = g(z_5)$$

is an invariant for this circuit. □

Usually, this method generates many invariant clauses. However, there are more invariants that can be automatically generated from the predecessor state predicate. These methods are described in the next two subsections.

4.2 Method 2

The second method is to convert existential quantification into disjunction when the quantified variable has a finite domain. For example, if x_k has a domain of $v_1, v_2, ..., v_l$, then the function $f(x_k)$ can be written as $f(v_1) \vee f(v_2) \vee ... \vee f(v_l)$.

Note first that since $P_1[\mathbf{z}]$ may in general contain a conjunction of many clauses, a direct application of this method to $P_1[\mathbf{z}]$ would result in an expression size exponential in the number of bound variables. In order to avoid this, we first move the conjunction outside the scope of the quantifiers. This still results in a size increase which is linear in the size of the finite domain. Though efficient for Boolean variables, it can produce a sizable blow-up for other variables. Thus we apply Method 1 first in order to eliminate as many bound variables as possible.

By moving the conjunction outside the scope of the quantifiers, we get an approximation which is not equivalent to, but is implied by the predecessor state predicate. Call the result $P_1'[\mathbf{z}]$:

$$P_1'[\mathbf{z}] = \bigwedge_{i=1}^{n_1} [\exists \mathbf{w}, \mathbf{x}.\ z_i = \tau_i'[\mathbf{w}, \mathbf{x}, \mathbf{z}]]$$

We apply the finite-domain transformation described above to each of the clauses in the conjunction $P_1'[\mathbf{z}]$, replacing each clause with a disjunction (these disjunctive clauses are simplified to eliminate duplicate terms). Note that we can substitute for functions as well as variables which can be very advantageous: if we have a function $f(x)$ whose range is small, we can perform the transformation on $f(x)$ even if x has a very large domain. As in the first method, after performing these transformations, some of the clauses may no longer have bound variables. These clauses are then combined via conjunction to form $I_2[\mathbf{z}]$. Since $P_1'[\mathbf{z}] \Rightarrow I_2[\mathbf{z}]$ is valid, by transitivity $P[\mathbf{z}] \Rightarrow I_2[\mathbf{z}]$ is valid so that $Q_0[\mathbf{z}] \vee I_2[\mathbf{z}]$ is an invariant by Theorem 1.

In contrast to the first method, which generally produces invariants that are strong constraints on state variables, this method tends to produce weaker constraints which enumerate all the possible values of a state variable or all the possible relationships among several state variables. We illustrate by continuing with the same example.

Example 2 *At the end of the last example, we were left with the following residual expression* $P_1[\mathbf{z}]$*:*

$\exists \mathbf{w}, \mathbf{x}.\ [z_1 = x_1\ \wedge\ z_2 = w_1\ \wedge\ z_3 = (ite\ (x_2 = z_2)\ x_2\ z_5)\ \wedge\ z_5 = f(x_3)].$

We then distribute the quantifiers inside the conjunction to get:

$[\exists \mathbf{w}, \mathbf{x}.\ z_1 = x_1]\ \wedge\ [\exists \mathbf{w}, \mathbf{x}.\ z_2 = w_1]\ \wedge\ [\exists \mathbf{w}, \mathbf{x}.\ z_3 = (ite\ x_2\ w_2\ z_5)]$
$\wedge\ [\exists \mathbf{w}, \mathbf{x}.\ z_5 = f(x_3)].$

Now, suppose that the bound variable x_3 *has the domain* $\{v_1, v_2, v_3\}$ *and that it is the only variable with a finite domain. We make use of this fact to get:*

$\cdots\ \wedge\ [z_5 = f(v_1)\ \vee\ z_5 = f(v_2)\ \vee\ z_5 = f(v_3)].$

Using the same initial state predicate as before, we have the following new invariant:

$\mathbf{z} = \mathbf{q_0}\ \vee\ z_5 = f(v_1)\ \vee\ z_5 = f(v_2)\ \vee\ z_5 = f(v_3).$ □

4.3 Method 3

The third and final method consists of two transformations that can be used to eliminate expressions containing bound variables within ite expressions. The first is much like Method 2: in particular, the expression $z_i = (ite\ \alpha\ \beta\ \gamma)$ is transformed to $(z_i = \beta) \vee (z_i = \gamma)$ when α contains a bound variable, and β and γ do not (the difference from Method 2 is that we do not explicitly enumerate the possible values of α). The transformed expression is implied by the original, so it can be used in an invariant.

The other transformation applies when α is of the form $a = b$: a may be substituted for b in β since β can only affect the truth of the expression if $a = b$. This is useful when a and γ contain no quantified variables, and b contains the only bound variables in β. A dual transformation can be performed if α is of the form $a \neq b$. In this case, we can substitute a for b in γ since γ can only affect the truth of the expression if it is not the case that $a \neq b$ (which means $a = b$).

Example 3 *We continue with the residual expression left from the end of Example 2:*
$[\exists \mathbf{w}, \mathbf{x}.\ z_1 = x_1] \ \wedge\ [\exists \mathbf{w}, \mathbf{x}.\ z_2 = w_1] \ \wedge\ [\exists \mathbf{w}, \mathbf{x}.\ z_3 = (ite\ (x_2 = z_2)\ x_2\ z_5)]$.
We transform the last clause to get:
$\cdots \ \wedge\ [z_3 = z_2 \ \vee\ z_3 = z_5]$.
Thus we have the following invariant: $\mathbf{z} = \mathbf{q_0} \ \vee\ z_3 = z_2 \ \vee\ z_3 = z_5$. □

4.4 Invariant extraction procedure

The three methods we have presented are most effective when used together. As mentioned, Method 1 is applied first, and Methods 2 and 3 are then applied to the remaining predicate. It is important to apply Method 1 first because the bound variables eliminated by Method 1 may significantly reduce the blow-up caused by Method 2. Similarly, it is important to apply Method 3 last because Method 3 requires choices to be made based on which expressions contain bound variables and the other two methods may be able to eliminate some of those bound variables. In our example we were able to produce three invariants. The individual invariants can then be combined. For our example, the strongest invariant we can generate is thus:

$$\begin{aligned}\mathbf{z} = \mathbf{q_0} \ \vee\ [&z_4 = g(z_5) \ \wedge\ (z_5 = f(v_1) \ \vee\ z_5 = f(v_2) \ \vee\ z_5 = f(v_3)) \\ &\wedge (z_3 = z_2 \ \vee\ z_3 = z_5)].\end{aligned}$$

5 Experimental Results

In this section, we present the results of applying our three methods for finding invariants to the Stanford Protocol Processor (PP). We first briefly describe the PP architecture.

5.1 Protocol Processor

PP is a pipelined microprocessor with a dual pipeline. The two sides (A and B) of the pipeline have similar functionality except for control and memory instructions. Control instructions are always executed on the A-side, while memory instructions are executed on the B-side. The processor has one delay slot for branch and load instructions, separate instruction and data caches which are two-way associative, and five stages in its pipeline which are very similar to the stages in the DLX pipeline [5]. PP does not have hardware interlocks, relying on compiler scheduling to avoid dependency problems. It does allow a load to bypass a store if they are accessing different cache lines, but if they access the same cache line, PP stalls the whole pipeline for two cycles to resolve the conflict. There is also a write back buffer to speed up cache replacement. All these architecture features are very difficult to verify all at once. Thus, we decided to divide the verification into two parts: the pipeline controller and the cache controller.

5.2 PP Pipeline Controller

We first wrote descriptions of the specification and implementation in our behavior description language. The specification was translated from the PP instruction set architecture specification [16], and the implementation was translated from the PP Verilog description. Our PP implementation has nine instructions: ALU, ALUI, B, BR, J, JR, JAL, LD, SD. The ALU and ALUI instructions abstract all of the ALU instructions (non-immediate and immediate respectively). We believe that these nine instructions represent the main features of PP.

Controller	Manual Method	Automatic Method
Pipeline	7	6
Cache	72	28

Table 1. Number of invariants needed for manual method compared to those generated by the automatic methods in the verification of PP controllers.

A manual top-down approach to verifying the controller required the addition of seven strengthening invariants (see table 1). Our automatic methods found six of these invariants. Although the automatic method did not find all of the needed invariants, it only took a few hours, while the manual method took several weeks.

5.3 PP Cache Controller

We next turned our attention to the PP cache controller. The cache controller has three finite state machines: the conflict FSM, the miss FSM, and the replacement FSM. Because of complicated interactions among these three state machines, the verification of the cache controller is much more difficult than the verification of the pipeline controller. Thus, larger and more complicated invariants are required. Using a top-down manual approach, we found that 72 invariants were necessary to verify correctness. This took several months. The automatic method found 28 of these invariants (in fact, the automatic method produced 61 simpler invariants, which implied 28 out of 72 of the manually-discovered invariants). Thus, we still needed to provide 44 invariants by hand. Still, the automatic invariants are generated in just a few hours, so the overall

Manual Method	Automatic Method
($\Leftrightarrow$ DEXTREAD-S2 (= EXTSTATE-S1 @EXT_WAITOK2))	($\Leftrightarrow$ DEXTREAD-S2 (= EXTSTATE-S1 @EXT_WAITOK2))
($\Leftrightarrow$ DEXTREQ-S2 (or (= FLUSHSTATE-S2 @F_WAITOK) (= EXTSTATE-S2 @EXT_WAITOK2)))	($\Leftrightarrow$ DEXTREQ-S2 (or (= FLUSHSTATE-S2 @F_WAITOK) DEXTREAD-S2)
(and (ite (and (or INSTRISLOAD-S2M INSTRISSTORE-S2M ($\neq$ CONFSTATE-S2 @CONF_IDLE)) (= EXTSTATE-S2 @EXT_IDLE) DTAG1-RW1-S2) DTAGREAD-S2M true) ...)	(or ($\Leftrightarrow$ (or (and ($\neq$ CONFSTATE-S2 @CONF_IDLE) DTAG1-RW1-S2) (= EXTSTATE-S2 @EXT_PROBE) DEXTREAD-S2) DTAGREAD-S2M) ($\Leftrightarrow$ (or DTAG1-RW1-S2 (= EXTSTATE-S2 @EXT_PROBE) DEXTREAD-S2) DTAGREAD-S2M))

Table 2. A comparison of manually and automatically generated invariants in the PP cache controller verification. $\Leftrightarrow$ denotes "if and only if", *ite* denotes "if then else", and @ indicates a symbolic constant. Variable names are in all capital letters while logic operations use lower-case letters. The first pair of invariants show that DEXTREAD-S2 is true if and only if the variable EXTSTATE-S1 is in the @EXT_WAITOK2 state. The automatically and manually generated invariants are identical. The second pair of invariants constrain the variable DEXTREQ-S2. The two invariants are in different forms, but are equivalent. The third pair of invariants are very different. Each of them constrain the variable DTAGREAD-S2M, but the manual method includes this as part of a larger and more complicated constraint.

time required for verification is significantly reduced. Some typical invariants in this verification are listed in table 2.

The design of the memory system is qualitatively different from the design of the CPU, in that the memory system is much more *control-intensive*. Indeed, it has three finite-state machines that are explicitly implemented in the description. Many of the necessary invariants deal with characterizing the different reachable combinations of states from these machines. The method described above is ill-suited for finding these combinations.

There exist several extremely efficient ways of finding the reachable state combinations of a collection of machines already (indeed, this is the central task in finite-state verification). An obvious approach, which we intend to explore, is to separate out the finite state machines from a description, analyze the state combinations separately, then combine the results with invariants extracted by the method described above. We believe this approach could find most of the invariants required for the PP memory system.

6 Conclusions and Future Work

We have presented some simple heuristics for automatic invariant discovery in processor verification. Surprisingly, these simple methods are effective in discovering many of the invariants required to verify a real microprocessor. Although we have focused on the Burch and Dill verification paradigm, these techniques should be useful in any methodology in which a syntactic description of the state transition function can be obtained. Combining the method with more powerful tools and other ways of discovering invariants should produce even better results.

An obvious generalization of our methods is to simulate for more than one cycle and use the multi-cycle next state function to generate invariants. Our experimentation to date has shown that the results are too complex to be useful, but the idea may prove to be useful on future designs.

Acknowledgments

This research was sponsored by ARPA contract ARMY DABT63-95-C-0049-P00002, and the third author is supported by a National Defense Science and Engineering Graduate Fellowship. The authors also wish to thank Robert Jones for his help and advice in preparing the final revision of the paper.

References

1. J. Burch and D. Dill, “Symbolic Verification of Pipelined Microprocessor Control”, 6th Computer Aided Verification, 1994.
2. J. Burch, E. Clarke, K. McMillan, D. Dill, and L. Hwang, “Symbolic Model Checking: 10^{20} States and Beyond”, 5th Annual IEEE Symposium on Logic In Computer Science, 1990.
3. A. Cohn, “A Proof of Correctness of the VIPER Microprocessors: The First Level”, In VLSI Specification, Verification and Synthesis, 1988.
4. D. Cyrluk, “Microprocessor Verification in PVS: A Methodology and Simple Example”, Technical Report SRI-CSL-93-12, SRI Computer Science Laboratory, Dec. 1993.
5. J. Hennessy and D. Patterson, “Computer Architecture: A Quantitative Approach”, Morgan Kaufmann, 1990.
6. W. Hunt, Jr., “Microprocessor Design Verification”, Journal of Automated Reasoning 5: p429-460, 1989.
7. R. Jones, D. Dill and J. Burch, “Efficient Validity Checking for Processor Verification”, IEEE/ACM International Conference on Computer Aided Design, 1995.
8. J. Joyce, G. Birtwistle, and M. Gordon, “Proving a Computer Correct in Higher Order Logic”, Technical Report 100, Computer Lab., University of Cambridge, 1986.
9. M. Langevin and E. Cerny, “Verification of Processor-like Circuits”, In Advanced Research Workshop on Correct Hardware Design Methodologies, June 1991.
10. Z. Manna, et al., “STeP: the Stanford Temporal Prover”, Technique Report, STAN-CS-TR-94, Computer Science Department, Stanford, 1994.
11. Z. Manna and R. Waldinger, “The Deductive Foundations of Computer Programming”, Addison Wesley, 1993.
12. Z. Manna and A. Pnueli, “Temporal Verification of Reactive Systems: Safety”, Springer-Verlag, 1995.

13. S. Bensalem, Y. Lakhnech, and H. Saidi, “Powerful Techniques for the Automatic Generation of Invariants”, To appear in CAV96.
14. A. Roscoe, “Occam in the Specification and Verification of Microprocessors”, ACM Trans. Prog. Lang. Syst., 1(2):245-257, Oct. 1979.
15. J. Saxe, S. Garland, J. Guttag, and J. Horning, “Using Transformations and Verification in Circuit Design”, Technical Report 78, DEC System Research Center, Sept. 1991.
16. R. Simoni, “PP Instruction Set Architecture Specification”, version 1.7, Stanford FLASH group, 1995
17. M. Srivas and M. Brickford, “Formal Verification of a Pipelined Microprocessor”, IEEE Software, 7(5):52-64, Sept. 1990.
18. C. Barrett, D. Dill, and J. Levitt, “Validity Checking for Combinations of Theories with Equality”, To appear in FMCAD, 1996.