Presented at the first IEEE International Conference on Formal Engineering Methods (ICFEM '97), Hiroshima, Japan, November 1997, pp. 140–149.

Systematic Formal Verification of Interpreters*

David Cyrluk, John Rushby, Mandayam Srivas
Computer Science Laboratory
SRI International
Menlo Park CA 94025 USA
{Cyrluk|Rushby|Srivas}@csl.sri.com

Abstract

Formal methods have gained acceptance in the hardware field through a pragmatic approach that has succeeded in providing systematic, scalable, highly automated, and cost-effective treatments for certain stereotypical problems of practical importance. By identifying stereotypical problems, the effort required to develop effective formal methods has been amortized over many applications. We suggest that formal methods can achieve similar industrial success in selected software applications by following the same principles.

As illustration, we examine approaches to the stereotypical problem of interpreter correctness in the presence of timing differences between the specification and implementation interpreters. In hardware, this corresponds to the problem of verifying microprogrammed, pipelined, or superscalar processors, but it has wider applications to any system—hardware or software—that can be considered as an interpreter.

1. Introduction

While advocates of formal methods agonize at the slow takeup of formal methods in the software industry, every major microprocessor and CAD company in the USA has established a formal methods program and is applying mechanized formal methods to some elements of hardware design, on at least an experimental basis (see, for example, papers in recent CAV [1, 12] and FMCAD proceedings [20]). Some of the reasons for the greater acceptance of formal methods in hardware industry than in software were described recently by Dill and Rushby [11], and include the following.

*This work was supported by Arpa through USAF Rome Laboratory Contract No. F30602-96-C-0204, by NASA Langley Research Center under contract NAS1-20334, and by the National Science Foundation under contract CCR-9509931.

Effective Tools: The evolution of tools that can deliver practical benefits to hardware designers is the primary reason why formal methods are being adopted in this industry. These tools perform highly automated formal *calculations* at various levels of the hardware design hierarchy.

At the lowest level are tools that check Boolean equivalence of combinational circuits (this is useful for checking manual circuit optimizations). Techniques based on Ordered Binary Decision Diagrams (BDDs) [3] are able to check large circuits quite efficiently, and are now incorporated in industrial CAD tools.

At a higher level, hardware designs can often be represented as interacting finite state machines, and tools that systematically explore the combined state space can check that certain desired properties always hold, or that undesired circumstances never arise. Tools based on explicit state enumeration can explore many millions of states in a few hours; tools that represent the state space symbolically (using BDDs) can sometimes explore vast numbers of states (e.g., 10^{100}) in the same time, and can check richer properties (e.g., those that can be specified in a temporal logic, in which case the technique is called "temporal logic model checking") [14].

At the highest levels, or when very complex properties or very large (or infinite) state spaces are involved, highly automated theorem proving methods can be used to compare implementations with specifications. These theorem proving methods include decision procedures for propositional calculus, equality, linear arithmetic, and bitvectors, combined with rewriting, and induction [2, 8–10].

The focus is refutation rather than verification: Engineers are at least as interested in finding bugs early in the development process as in establishing their absence at its very end. Shortcuts can be taken when formal methods are used for finding bugs (refutation) rather than proving correctness (verification). For example, the number or sizes of modeled components can be drastically reduced (a process called "downscaling"), so that a directory-based cache-coherence protocol, say, can be examined in a model with just four processors, one cache line, and two data values. Such a downscaled description will still have many millions of states, but may be within reach of state exploration and model checking methods. These methods can check the reduced system *completely*; in contrast, simulation checks the full system very *in*completely. Both techniques find some bugs and miss others, but experience indicates that full exploration of a downscaled system finds bugs activated by complex scenarios that are seldom encountered in testing or simulation, but that do occur—usually at the worst time—in real use. Finite state methods not only identify the existence of bugs, but can generally exhibit scenarios that activate them, thereby simplifying interpretation of their output and increasing its utility.

Targeted techniques: Experience shows that control-dominated hardware circuits are much harder to debug than data paths. So effort has gone into developing formal verification techniques for protocols and controllers rather than for data paths. Targeting maximizes the potential payoff of formal methods by solving problems that are not handled by other means. Notice that the targeted problems often concern the *hardest* challenges in design: cache coherence, pipeline (and now superscalar) correctness, and floating point arithmetic.

Stereotypical applications: Many hardware devices share similar architectural elements: for example, most processors have pipelines and caches; most floating-point units use dividers based on the SRT method, and multipliers similar to the Wallace-tree method; and shared-memory multiprocessors all use directory-based cache-coherence algorithms. Consequently, effort spent in understanding how best to apply formal methods to one of these architectural elements can be amortized over several applications; in addition, several groups often attack similar problems, so that competition spurs progress and the best methods become widely promulgated.

Driven by real problems: Research priorities in formal methods for hardware are largely driven by their practical application to real, or realistic, problems. Such efforts frequently raise interesting new theoretical problems, as well as highlighting the need for improvements in tools.

Applying formal methods to real problems is also crucial for building credibility. There is now a long string of success stories from academia and industry where finite-state verification techniques have been applied to hardware and protocols. A couple of documented examples include cache-coherence bugs in FutureBus+ [5] (found using symbolic model checking with a version of CMU's SMV system), and formal verification of the microarchitecture and selected microcode of the Collins AAMP5 [21] and AAMP-FV avionics processors (using theorem proving with SRI's PVS system [15]).

Many of these factors that have contributed to interest in formal methods among hardware engineers can also be applied to software. In this paper, we focus on just one factor—stereotypical applications—and use verification of interpreters as an example that illustrates some of the practical issues that must be addressed.

In the rest of this paper, we first identify the significant issues in verifying interpreter correctness; we then introduce two approaches for undertaking such verifications and demonstrate them on a simple pipelined microprocessor. We then compare the two approaches and conclude by drawing some lessons for applying formal methods to software.

2. Interpreter Correctness: The Issues

Unlike traditional programs, systems that operate continuously cannot be specified using input/output relations. Such *reactive systems* include most hardware circuits, protocols, and many software systems, including operating systems and those that control physical plants such as airplanes and CD players. Current practice is to specify such systems as state transition systems.

In the form in which we are interested in it, the problem is to verify one system (the *implementation*) against another (the *specification*), where both systems can be characterized as interpreters that consume inputs supplied by their environments and undergo state changes in response. Some elements of the states of the two systems are visible to the environment. To make things concrete, the specification system could be a microprocessor described at the instruction set architecture (ISA) level—that is, at the level understood by a machine-code programmer—while the implementation system could be the same microprocessor described at the register transfer level (RTL)—that is, at the level of its microarchitectural building blocks (such as caches, pipeline registers, ALU, shifters, and microcode memory). The inputs to both levels of system description

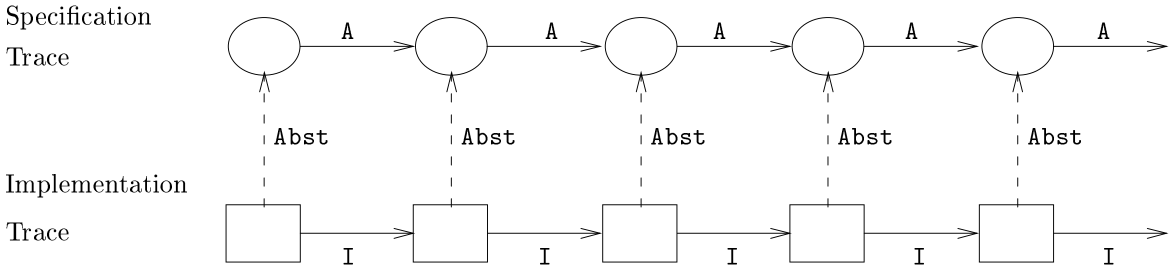

Figure 1. Trace Equivalence

are machine instructions, and the goal is to show that the state of the RTL description after processing some arbitrary stream of instructions is equivalent to that of the ISA description after processing the same stream, given equivalent starting states.

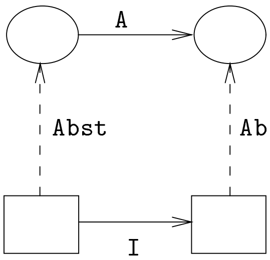

Figure 2. The Commutes Condition

The basic form of an interpreter correctness proof has been known for over 25 years [13]: both specification and implementation systems are specified as state machines, an *abstraction function* maps implementation states to corresponding specification states, and it is verified that each transition takes corresponding states to corresponding states (i.e., that the diagram of Figure 2 commutes, where `A` and `I` are the *next-state* functions for the specification and implementation machines, respectively, and `Abst` is the abstraction function). In order that every starting state in the specification have a corresponding implementation state, the abstraction function is required to be surjective. Induction then establishes that the *traces* (i.e., sequences of states) generated by the implementation system are a subset of (i.e., are allowed by) the traces of the specification machine (see diagram 1).

In practice, this simple and well understood approach needs substantial elaboration to deal with the complexities of real systems, and to allow automated proof strategies to work effectively. The complexities mostly arise from timing differences between the two levels of system description. Microprocessor verification, for example, faces the following sources of timing differences.

Microsequencing. Instruction execution is often broken down into several *microsteps* in the implementation machine. The exact number of steps may be fixed for each type of instruction, or it may be data-dependent (e.g., the number of microsteps in a `shift` instruction may depend on the number of bits to be shifted). Both these cases may be further complicated by the interposition of arbitrary numbers of microsteps to perform cache and stack management, or by delays due to memory latency. The consequence of these timing differences is that it is no longer obvious how to establish the correspondence between states in the specification and implementation traces, given that they no longer have the same number of elements.

Pipelining. Even though it takes several microsteps to execute a single instruction, modern processors attempt to complete one instruction per microstep by performing different steps of consecutive instructions in parallel—a process known as *pipelining.* This complicates the verification task because implementation states no longer have a simple correspondence to specification states: specification states represent points at which execution of one instruction has been completed, but its successor has not yet begun, whereas implementation states generally have several partially completed instructions in progress simultaneously. In addition, the pipeline introduces latency, so that although one instruction generally completes at each step, it takes several steps for an individual instruction to work its way down the pipe. This means that the abstraction function must generally be "skewed" in time, so that an implementation state will correspond a specification state some time units in the past. This issue is further complicated by pipeline stalls, bypass registers, and interrupts, all of which change the time skew.

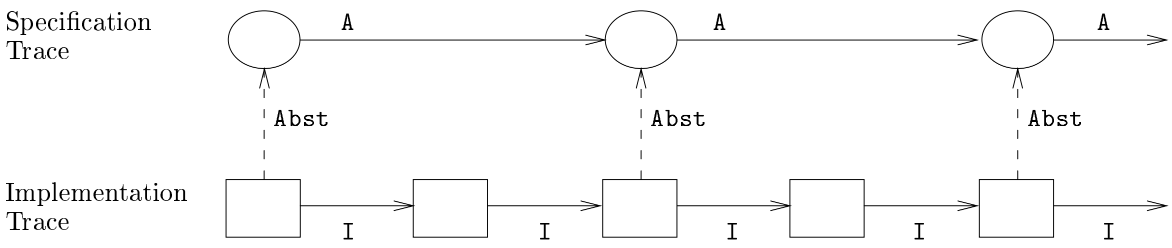

Figure 3. Timing Differences

Superscalar. Fast processors attempt to complete *more* than one instruction per microstep using multiple pipelines and other independent execution units. This further compounds the difficulty in establishing correspondence between the one-instruction-per-step model of instruction execution at the specification level and the highly variable timing behavior at the implementation level.

The basic interpreter-correctness methodology must be elaborated to overcome the difficulties introduced by timing differences between the specification and implementation levels. However, it is not enough to provide elaborations that are merely valid: the elaborated methods must lend themselves to efficient mechanization. There are two elements to efficient mechanization.

- Relatively little human guidance must be required.

  Interpreter correctness proofs of real-world artifacts such as microprocessors require billions of elementary deductive steps, so that step-by-step human guidance of an interactive proof checker is simply infeasible. Human intervention must be limited to establishing the overall proof strategy (e.g., treating each different opcode as a separate case) and, perhaps, guiding some small number (no more than half a dozen) key steps in the verification of each major case.

- The mechanization must exploit powerful automation.

  The theorem provers in modern verification systems derive their power mainly from the integration of rewriting with decision procedures [16]. Specifications and proof strategies must therefore be constructed to maximize the extent to which these capabilities can be exploited.

In the following section, we examine two approaches for dealing with timing differences between the implementation and specification machines.

3. Dealing with Timing Differences

There are two basic techniques for extending the simple interpreter-correctness methodology to deal with timing differences. We illustrate the ideas for the case, such as that shown in Figure 3, where the implementation machine runs slower than the specification machine (i.e., ignoring superscalar issues for the moment). The basic problem is that it can now take several implementation steps to accomplish a single specification step (e.g., two steps in the case of Figure 3) and this leads to intermediate implementation states that have no natural corresponding implementation states.

One technique is to ignore the intermediate states in implementation traces and to require that only certain states designated as "visible" are in correspondence with specification states. This is called the *visible state* approach. The other technique is to add extra "stuttering" states to specification traces so that all implementation states have corresponding specification states. This is called the *stuttering state* approach. The two techniques are described in the following sections.

3.1. The Visible State Approach

In this approach (which originated with Srivas and Bickford [19]) we distinguish two types of implementation states: those that are intended to correspond to a specification state are called *visible* states; the others are called non-visible states. This is illustrated in Figure 4, where non-visible states are shown dotted. For a microprogrammed machine, the visible states might be those where interpretation of a new machine instruction is about to start; for a pipelined machine, they might be those in which the pipeline is not stalled.

For verification, the next-state function for the implementation machine is replaced by a *next-visible-state* function that is constructed by combining the transitions from one visible state through zero or more non-visible states to the next visible state; in Figure 4 this function is shown as `II`.

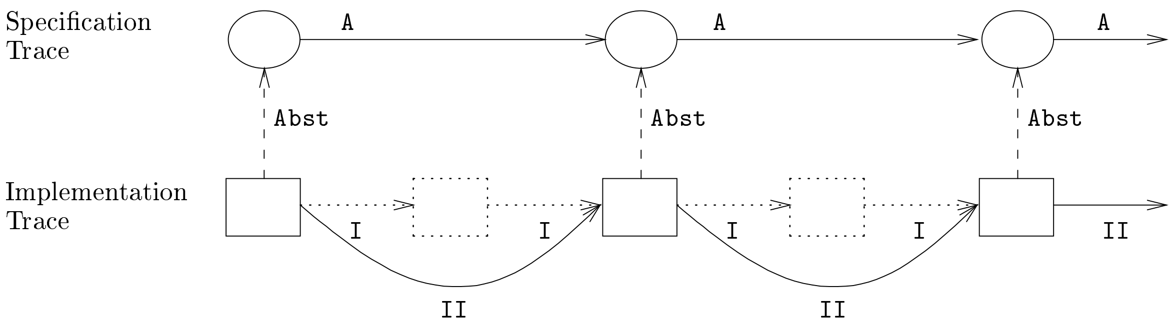

Figure 4. Visible Implementation States

One way to define the *next-visible-state* function is in terms of a predicate that characterizes the visible states: *next-visible-state* then simply iterates the next-state function of the implementation machine until a visible state is encountered. We need to be sure that the iteration cannot continue indefinitely (i.e., that there *is* a visible next state), so it is generally convenient to introduce a function *num-non-vis* which, given the current implementation state and input, returns the number of iterations required to reach the next visible state (in Figure 4 this function would return the constant 2). The *next-visible-state* function can then be defined to iterate the next-state function *num-non-vis* times. Defining the *num-non-vis* function requires extra work on the part of the specifier, but it has the additional benefit of greatly increasing the automation possible in the proof of the commutes condition.

3.2. The Stuttering Approach

Whereas the visible state approach brings the implementation and specification traces into alignment by "skipping over" the excess implementation states, the stuttering state approach adds new specification states so that each implementation state has a corresponding specification state. This approach is accomplished by modifying the specification next-state function to have a "do nothing" effect (to *stutter*) whenever the implementation machine is in an intermediate state (i.e., what was called a non-visible state in the previous approach).[1] The abstraction function also must be modified to indicate how an intermediate implementation state maps to a specification state.

The difficulty in this approach is that the modified specification next-state function must be defined either to perform its ordinary next-state function or to stutter, and the decision which is based on whether the *implementation* machine is in (or will transition to) an intermediate state. Since the abstraction function goes from implementation to specification states, and is not generally required to be injective, it is difficult to refer to implementation states when defining the modified specification next-state function. For this reason, the stuttering approach is often combined with a method for specifying state machines called the *explicit time* approach that allows both levels of description to refer to a common notion of "time."

Explicit Time

Traditionally, a next-state function takes a state and returns its successor; there is no explicit notion of "time" nor of the t'th state. Of course, one can identify "time" with the number of iterations of the next-state function: given a starting state and a sequence of inputs, iteration of the next-state function will generate a sequence of states in which it is possible to identify the t'th state. We can denote this state by $Q(t)$ and then write

$$Q(t+1) = \textit{next-state}(Q(t)),$$

where $Q(0)$ is some distinguished starting state.

We can then add stuttering to the specification machine as follows, where S and s denote the specification and implementation states, respectively, A is the specification next-state function,[2] and *stutter* is a predicate on implementation states that determines whether the specification machine should take a stuttering transition. (Typically, *stutter* is the negation of the *visible*

[1] Alternatively, the specification machine can be defined to stutter whenever the implementation machine transitions *to* an intermediate state; it is a matter of preference and convenience which definition is used.

[2] The instructions to be executed are held in memory, which is regarded as part of the state; hence the next-state functions simply take a state (and not a state and an input) as their arguments. Inputs are reintroduced in more elaborate treatments that deal with, for example, external interrupts.

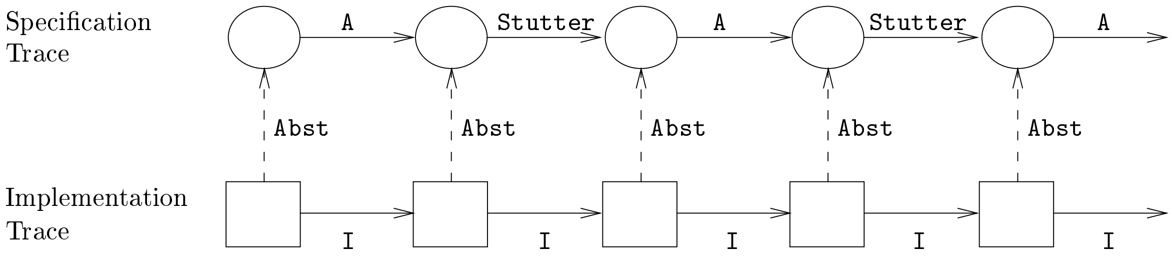

Figure 5. Stuttering the Specification Machine

predicate mentioned in the previous section.)

$S(t+1) = \texttt{IF}\ \mathit{stutter}(s(t))\ \texttt{THEN}\ S(t)\ \texttt{ELSE}\ A(S(t))$.

Whereas specifications using the visible state approach usually define the machine state as a record (whose components correspond to registers) and give a monolithic specification for the next-state function, specifications using the stuttering approach with explicit time usually specify the effect of the next-state function separately for each component of the state by means of equations that relate the value of the state component at time $t+1$ to the values of state components at time t. Derived functions such as *stutter* are specified similarly, so that the term $\mathit{stutter}(s(t))$ above would be replaced by $\mathit{stutter}(t)$.

4. An Example

We now illustrate how the two different approaches can be applied to a small but nontrivial example: a simplified pipelined microprocessor. This example is due to Saxe, Garland, Guttag and Horning [18], who performed the original verification using the stuttering state and explicit time approach. The example, and its verification in PVS using both approaches, are described in detail in a technical report [6].

Figure 6 shows the non-pipelined version of the microprocessor; this description is used as the specification machine; Figure 7 shows the pipelined implementation of the microprocessor. Corresponding registers in the two figures are labeled `SXX` for specification register `XX` and `IXX` for the implementation register (or `IXX`n for n = 1, 2,... where the implementation has multiple registers).

Two types of instructions are distinguished in this microprocessor: normal instructions and branch instructions. The decision whether a branch instruction should cause execution to jump to the branch target (BT) address rather than merely increment the program counter (PCtr) like a normal instruction is performed by the `zero?` test on the output of the ALU

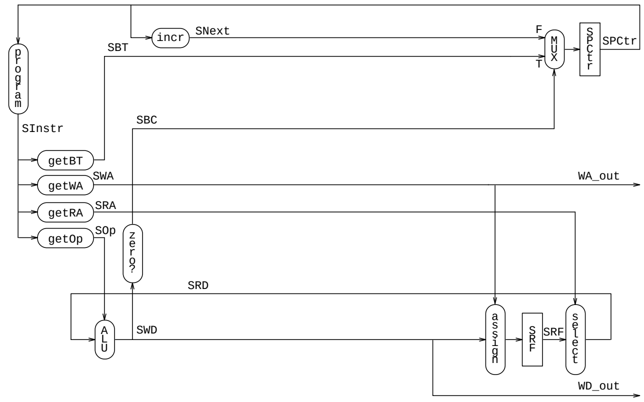

Figure 6. Non-pipelined microprocessor

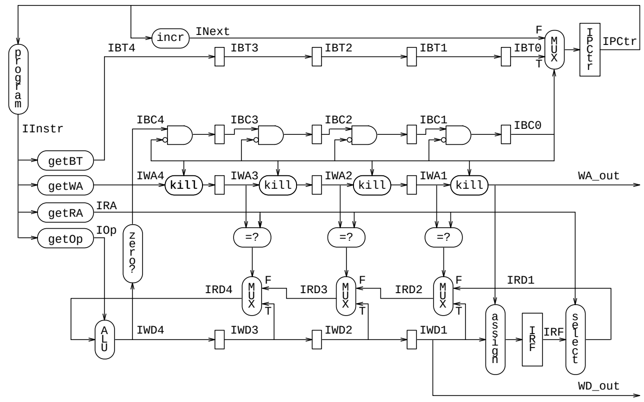

Figure 7. Pipelined microprocessor

(wire `SBC` in the specification, and wires `IBC`n in the implementation). Instructions may read and write data in the register file (RF); the addresses read and written are denoted `SRA` and `SWA`, respectively, in the specification, and `IRA` and `IWA`n in the implementation, and the corresponding data values are denoted `SRD` and `SWD` in the specification and `IRD`n and `IWD`n in the implementation.

The implementation machine differs from the specification machine in having a four-stage pipeline that uses four registers to implement each of the `SBC`, `SBT`, `SWA`, and `SRD` specification registers. The registers along the `IBCn` path delay the decision on whether to branch by four cycles, while those along the `IBTn` path delay the target of the branch for four cycles also.

When a branch instruction in the final stage of the pipeline causes a jump to the branch target, it is necessary to abort the three partially completed instructions in the other pipeline stages because these were loaded from addresses that were computed on the assumption that the branch would not be taken. The `IWAn` path delays writes to the register file and allows them to be aborted if a branch is taken. Aborts are accomplished using the `kill` element which returns the address `null` if its conditional input is *true*, otherwise it passes its input address through unchanged. The register file is unchanged when data is written to the `null` address.

The `IWDn` path with the corresponding multiplexors (`muxs`) and comparators (`=?`) implements the register bypass logic. If an instruction needs the value of a register that a previous instruction has written, but which has not yet been committed to the register file (due to the pipeline delay), then this logic forwards the value of that register directly back to the ALU.

4.1. Visible State Approach

To apply the visible state approach, we need to identify the visible states and provide specifications for the *num-non-vis* and abstraction functions. Visible states are those in which the pipeline is not *stalled*: that is, not engaged in aborting partially completed instructions due to a branch instruction. The pipeline is stalled when one of `IBC0` to `IBC3` is *true*, so we can define

```
visible(s) = NOT IBC0(s) AND NOT IBC1(s)
                 AND NOT IBC2(s) AND NOT IBC3(s)
```

where `s` is an implementation state. If `IBC4` is *true* in a visible state, then the pipeline will stall for the next four cycles, so that we can define

```
num-non-vis(vs) = IF IBC4(vs) THEN 5 ELSE 1
```

where `vs` is a visible implementation state. Now the *next-visible-state* function can be defined as

$$\texttt{next-visible-state(vs) = I}^{\texttt{num-non-vis(vs)}}\texttt{(vs)}$$

where `I` is the original implementation next-state function and the superscript denotes iteration that many times.

Finally, we can specify the abstraction function that maps a visible implementation state to its corresponding specification state. The specification machine state is defined by the values of the program counter, `SPCtr`, and the register file, `SRF`. The abstraction function, `Abst` must specify how `SPCtr` and `SRF` are obtained from a visible implementation state. This is accomplished by the following two equations.

$$\texttt{SPCtr(Abst(vs)) = IPCtr(vs)}$$
$$\texttt{SRF(Abst(vs)) = IRF(I}^3\texttt{(vs))}$$

Informally, the abstraction mapping states that the program counters of the two machines are updated at the same time but, due to pipeline delays, the register file of the specification machine corresponds to the register file of the implementation machine three (not necessarily visible) states later.

The Proof of Correctness

In addition to establishing surjectivity of the `Abst` function, verification of correctness of the implementation machine requires discharging the following two proof obligations.

Pipeline Invariant: The `next-visible-state` function is closed on visible states:

$$\texttt{visible(vs)} \supset \texttt{visible(next-visible-state(vs))}$$

Proof. Expanding the definition of `next-visible-state`, we obtain

$$\texttt{visible(vs)} \supset \texttt{visible(I}^{\texttt{num-non-vis(vs)}}\texttt{(vs))}$$

and the proof then requires case analysis based on the expansion of `num-non-vis`, followed by rewriting to replace functions with their definitions. The need for a pipeline invariant was first recognized by Srivas and Bickford [19]; as well as being an independent proof obligation, it is needed in the proof of the commutes condition.

Commutes Condition: This is the property corresponding to Figure 1.

```
A(Abst(vs)) = Abst(next-visible-state(vs))
```

Proof. The state of the specification machine has two components: the program counter `SPCtr` and the register file `SRF`, so this proof obligation divides into two cases. Both expand the `Abst` function, then the `next-visible-state` and `num-non-vis` functions. The latter suggests a further case-split on the value of `IBC4`. The case corresponding to the program counter is completed using rewriting to expand definitions, but the register file needs the following lemma that relates data read from the register file in the two machine descriptions.

Lemma: `IRD4(vs) = SRD(Abst(vs))`

This is proved by case analysis on the possible values of `IRD4`, which depends on whether internal forwarding is needed or not (that is, on whether `IRA` equals `IWA1`, `IWA2`, `IWA3`, or none of these).

4.2. Stuttering State Approach

Rather than give all the details of the example verification using the stuttering state approach, we simply highlight where it differs from the visible state approach presented in the previous section (see [6,18] for the details).

To relate the two machine descriptions, we need to define when the specification machine stutters, and the abstraction mapping from the implementation machine to the specification machine. The specification machine stutters exactly when the implementation machine goes through non-visible states. That is

```
stutter(t) = IBC0(t) OR IBC1(t)
                  OR IBC2(t) OR IBC3(t)
```

where `t` refers to "time" (i.e., steps) in the implementation machine. Observe that this is the negation of the definition of `visible` defined in the previous section.

The abstraction mapping is similar to that given in the previous section, but needs to map the non-visible as well as the visible implementation states into a corresponding specification state.

The Proof of Correctness

Because *every* implementation state is related to a specification state, there is no need to prove a pipeline invariant in this approach. The proof of the commutes condition is achieved by symbolically executing one step in the implementation machine and performing case analysis on the conditions that determine the next state. Symbolic execution uses the equations that make up the implementation machine and those that define the abstraction mapping as rewrite rules. Whereas the proof in the visible state approach was broken into cases depending on whether the instruction is a branch instruction, in the stuttering approach it breaks into cases for each different stuttering transition.

5. Comparing the Two Approaches

In this section we compare the two approaches using the pipeline example as a reference point. Both approaches have also been applied to much more substantial examples in research [4,21] and industry, including some state-of-the-art processor designs. We consider naturalness of specification, and ease of proof.

5.1. Specification

Both approaches allow very straightforward encodings of traditional RTL specifications given as diagrams or expressed in (the synthesis-level cycle-based subsets of) hardware description languages such as Verilog and VHDL. It should be possible to automatically translate such circuit descriptions into the specification styles used in these approaches.

The specifications are similar in the two approaches, except that in the stuttering approach the machine state includes the wires, not just the registers. This in turn requires a more complete abstraction mapping between the implementation machine and specification machine. In addition, the stuttering approach requires modifications to the specification machine to add the stuttering transitions.

The visible state approach, on the other hand, does not require any modifications to the specification (though it does require addition of the *num-next-visible* function). Because of this, however, it suffers certain other difficulties. The most important of these is that external inputs (such as interrupts) must somehow be consumed at the same rate in the two machines. (As there were no external inputs in our example, it did not manifest this problem.) There are several ways to arrange this: one is to have the specification machine consume the same number of inputs in each step as all the corresponding implementation steps (this is the approach taken in [19]). However, this approach, and its alternatives, run into a problem previously encountered in the stuttering approach: the specification machine must refer to the state of the implementation machine. The preferred way to deal with this in the visible state approach is to explicitly modify the specification machine to contain as much information as is needed to completely specify its functional behavior. If this is done, there is no need for the specification machine to refer to the implementation state and thus there is no need to introduce an explicit time variable as is used in the stuttering state approach.

5.2. Proof

Even though the underlying frameworks, and thus the ultimate statements of correctness, are different for the two approaches, the actual proof obligations are very similar and require similar interactive guidance to direct the case analysis. In both cases, this was accomplished by formulating the lemma that appeared in the proof of the Commutes Condition in Section 4.1.

Despite these similarities, however, the automation and efficiency of the two proofs are different: the stuttering approach requires more case analysis, while the visible state approach requires more rewriting. Overall, we found the total amount of CPU time required to prove the microprocessor correct (in PVS) using the stuttering approach was considerably less than for the visible state approach (the times involved are on the order of a few minutes on a slow Sun workstation). The following analysis suggests why this is so.

In the stuttering approach, only one implementation next state function is expanded for each branch in the proof. However, in the *visible state* approach, there are many non-visible next state transitions for one visible next state transition. In the cases where the instruction is a branch instruction, expanding the `next-visible-state` function requires expanding the `I` function five times. In addition, for the cases involving the register file, the `I` function must be expanded an additional three times (from the relationship stated in the `Abst` function). Thus, in the worst case, the `I` function must be expanded eight times. This involves considerable rewriting and results in large terms in the proof. On the other hand, in the stuttering approach, the proof of correctness must take into account the relationship between the non-visible implementation states and their corresponding stuttering specification states and this adds additional cases to the proof.

One other drawback to the stuttering approach is in dealing with termination of rewriting. In the visible state approach, rewriting terminates when there are no more `I` functions to rewrite. Since we are always dealing with an explicit number of state transitions this must always terminate. In the stuttering approach using explicit time, there is no monolithic next-state function, but rather a set of equations relating state variables at time $t+1$ with the state variables at time t. Thus, without intelligent control on rewriting we could get an infinite number terms involving $t-1, t-2\ldots$ PVS is able to avoid infinite rewriting by rewriting from $t+1$ to t only when it can prove that t is a natural number (i.e., $t \geq 0$).

5.3. Further Variations

The two approaches that we have described can each be extended in two further directions: one is to cope with additional complexities, such as that represented by superscalar processors, the other is to increase the automation that can be achieved.

One way to extend to superscalar machines (those that execute more than one specification instruction in each implementation step) is to provide a *num-abs-steps* function that takes an implementation state and returns the number of specification steps that the specification machine executes in the next implementation cycle (for an n-pipeline superscalar machine, this number will be n or less). The stuttering approach accommodates this extension by allowing the *num-abs-steps* function to return zero, in which case the machine stutters. The visible state approach does not allow *num-abs-steps* to return zero, instead it iterates the implementation next-state function enough times so that at least one specification step is taken. Verification of a simple superscalar machine is described in [7].

An alternative approach that seeks to provide greater automation was introduced by Burch and Dill [4]. Their method can be seen as a modification of the stuttering approach that avoids use of explicit time by modifying the commutes condition to take stuttering into account.

Commutes_normal:
$\forall s : \neg\texttt{stutter}(s) \supset \texttt{Abst}(\texttt{I}(s)) = \texttt{A}(\texttt{Abst}(s))$

Commutes_stutter:
$\forall s : \texttt{stutter}(s) \supset \texttt{Abst}(\texttt{I}(s)) = \texttt{Abst}(s)$

These two conditions can be combined into one and extended to superscalar machines using the `num-abs-steps` function mentioned earlier. Note that it is also necessary to prove that the machine does not stutter indefinitely.

A general difficulty in using stuttering approaches is that the abstraction mapping can become hard to specify because a stuttering step might update some visible state elements and not others. For example, in a microcoded machine the program counter might be updated near the beginning of the microcode sequence, but the register file might not be updated until the end, and the accumulator some time in-between.

Burch and Dill overcome this difficulty by introducing the notion of a "stall" mode that can "flush" the state of the implementation machine. In stall mode, the machine runs without consuming further inputs or instructions and gradually moves ("flushes") information from internal to visible state components (as if it were executing a series of `noop` instructions). The abstract state corresponding to any implementation state (including non-visible states) can be obtained by flushing the implementation machine until all its internal information has made its way to the visible state-components. In the case when there is no pipelining this roughly corresponds to running the machine until the next visible state is reached. In the case of pipelining the machine needs to be run further so that the contents of the pipeline are flushed into the register file. The attraction of this method is that it essentially automates construction of the abstraction function.

6. Conclusions and Relation to Software

Many verification problems can be formulated in terms of interpreter correctness and verified using the approaches described in the previous sections. Although we couched the discussion in terms of microprocessors, the same issues arise whenever the specification and implementation execute different numbers of steps—as, for example, when the implementation must execute several small programs to simulate the effect of a specification step, or when the implementation can perform several steps in parallel.

By making explicit the representational choices that are implicit in treatments of pipeline correctness such as those of [18] and [19], and by casting them in a more abstract framework, we hope to have facilitated comparison between the approaches and to have made them available to a broader audience, including that concerned with verification of software artifacts.

The larger point we wish to stress is that hardware verification has gained acceptance mainly by working out the practical details involved in providing systematic, scalable, highly automated, and cost-effective treatments for stereotypical problems of practical importance. We believe that formal methods can achieve similar success in selected software applications by following the same principles. Conversely, we encourage researchers in formal methods for software to consider industrial hardware applications. As processors become more complex, it is likely that their verification can benefit from some of the methods developed for software—for example, out-of-order execution and register renaming require complex data structures that may best be handled by approaches based on refinement.

Acknowledgements

Natarajan Shankar performed the initial PVS verification of the microprocessor example using the stuttering approach.

References

[1] R. Alur and T. A. Henzinger, editors. *Computer-Aided Verification, CAV '96*, volume 1102 of *Lecture Notes in Computer Science*, New Brunswick, NJ, July/Aug. 1996. Springer-Verlag.

[2] C. Barrett, D. Dill, and J. Levitt. Validity checking for combinations of theories with equality. In Srivas and Camilleri [20], pages 187–201.

[3] R. E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, Sept. 1992.

[4] J. R. Burch and D. L. Dill. Automatic verification of pipelined microprocessor control. In D. Dill, editor, *Computer-Aided Verification, CAV '94*, volume 818 of *Lecture Notes in Computer Science*, pages 68–80, Stanford, CA, June 1994. Springer-Verlag.

[5] E. M. Clarke, O. Grumberg, H. Haraishi, S. Jha, D. E. Long, K. L. McMillan, and L. A. Ness. Verification of the Futurebus+ cache coherence protocol. *Formal Methods in System Design*, 6(2):217–232, Mar. 1995.

[6] D. Cyrluk. Microprocessor verification in PVS: A methodology and simple example. Technical Report SRI-CSL-93-12, Computer Science Laboratory, SRI International, Menlo Park, CA, Dec. 1993.

[7] D. Cyrluk. Inverting the abstraction mapping: A methodology for hardware verification. In Srivas and Camilleri [20], pages 172–186.

[8] D. Cyrluk, P. Lincoln, and N. Shankar. On Shostak's decision procedure for combinations of theories. In M. A. McRobbie and J. K. Slaney, editors, *Automated Deduction—CADE-13*, volume 1104 of *Lecture Notes in Artificial Intelligence*, pages 463–477, New Brunswick, NJ, July/Aug. 1996. Springer-Verlag.

[9] D. Cyrluk, S. Rajan, N. Shankar, and M. K. Srivas. Effective theorem proving for hardware verification. In R. Kumar and T. Kropf, editors, *Theorem Provers in Circuit Design (TPCD '94)*, volume 910 of *Lecture Notes in Computer Science*, pages 203–222, Bad Herrenalb, Germany, Sept. 1994. Springer-Verlag.

[10] D. Cyrluk, H. Rueß, and O. Möller. An efficient decision procedure for the theory of fixed-sized bit-vectors. In Grumberg [12], pages 60–71.

[11] D. L. Dill and J. Rushby. Acceptance of formal methods: Lessons from hardware design. *IEEE Computer*, 29(4):23–24, Apr. 1996. Part of [17].

[12] O. Grumberg, editor. *Computer-Aided Verification, CAV '97*, volume 1254 of *Lecture Notes in Computer Science*, Haifa, Israel, June 1997. Springer-Verlag.

[13] C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1:271–281, 1972.

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

[15] S. Owre, J. Rushby, N. Shankar, and F. von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, Feb. 1995.

[16] J. Rushby. Automated deduction and formal methods. In Alur and Henzinger [1], pages 169–183.

[17] H. Saiedian (Ed.). An invitation to formal methods. *IEEE Computer*, 29(4):16–30, Apr. 1996. A "roundtable" of short articles by several authors.

[18] J. B. Saxe, S. J. Garland, J. V. Guttag, and J. J. Horning. Using transformations and verification in circuit design. *Formal Methods in System Design*, 4(1):181–210, 1994.

[19] M. Srivas and M. Bickford. Formal verification of a pipelined microprocessor. *IEEE Software*, 7(5):52–64, Sept. 1990.

[20] M. Srivas and A. Camilleri, editors. *Formal Methods in Computer-Aided Design (FMCAD '96)*, volume 1166 of *Lecture Notes in Computer Science*, Palo Alto, CA, Nov. 1996. Springer-Verlag.

[21] M. K. Srivas and S. P. Miller. Formal verification of the AAMP5 microprocessor. In M. G. Hinchey and J. P. Bowen, editors, *Applications of Formal Methods*, Prentice Hall International Series in Computer Science, chapter 7, pages 125–180. Prentice Hall, Hemel Hempstead, UK, 1995.