A Theory of Generic Interpreters

Phillip J. Windley

Laboratory for Applied Logic
Department of Computer Science
University of Idaho
Moscow ID 83843 USA

Abstract. We present an abstract theory of interpreters. Interpreters are models of computation that are specifically designed for use as templates in computer system specification and verification. The generic interpreter theory contains an abstract representation which serves as an interface to the theory and as a guide to specification. A set of theory obligations ensure that the theory is being used correctly and provide a guide to system verification. The generic interpreter theory provides a methodology for deriving important definitions and lemmas that were previously obtained in a largely ad hoc fashion. Many of the complex data and temporal abstractions are done in the abstract theory and need not be redone when the theory is used.

1 Introduction.

The formal specification and verification of microprocessors has received much attention. Indeed, several verified microprocessors have been presented in the literature. This paper presents a model, common to all of them, that can be used to guide future work in this area. The model defines an abstract microprocessor specification (called a generic interpreter) and proves important theorems about it.

We have formalized the interpreter model in the HOL theorem proving system [Gor88]. The formal model can be instantiated inside the system and serves as a framework for writing microprocessor specifications and verifying them. This framework clearly states what definitions must be made to specify the microprocessor and which lemmas must be established to complete the verification. After the user has defined the components of the microprocessor and proven the necessary lemmas about them, individual theorems from the abstract theory can be instantiated to provide concrete theorems about the microprocessor being verified.

The model that we have defined has proven useful in specifying and verifying several microprocessors [Win90a, Aro90, Coe92]. The model is not, however, limited to microprocessors. Recent work by the author has shown that the model can be used in specifying other hardware devices as well [Win91].

The model we have defined differs from other formal descriptions of state machines (such as Loewenstein's model in [Low89]) by including in the formalization the data and temporal abstractions that are important in specifying and verifying microprocessors.

2 Formal Microprocessor Modeling.

There have been numerous efforts to formally model microprocessors. The best known of these include Jeff Joyce's Tamarack microprocessor [Joy89], Warren Hunt's FM8501 microprocessor [Hun87], and Avra Cohn's VIPER microprocessor [Coh88]. Tamarack is a simple microprocessor with only 8 instructions. FM8501 is larger (roughly the size of a PDP-11), but has not been implemented (a 32–bit version is currently being verified and implemented by Hunt *et al* [Hun89]).

The specifications for the microprocessors mentioned above appear very different on the surface; in fact, the specification of FM8501 is even in a different language than the specifications of Tamarack and VIPER. On closer inspection, however, we find that each of them (as well as many others) use the same implicit behavioral model. In general, the model uses a state transition system to describe the microprocessor. We call this model an interpreter. The essence of verification is to relate mathematical models at different levels of abstraction.

The rest of this section gives a mathematical definition of the interpreter model and shows how two interpreters are related. In the discussion that follows, and for the rest of the paper, we speak of the "abstract level" and "concrete level," but keep in mind that these terms are relative; as we move up and down a hierarchy of interpreters, what we call "abstract" at one level will be termed "concrete" with respect to the level above it. As a matter of convention, we will decorate variables representing the concrete level with primes throughout the rest of the paper.

2.1 Interpreters.

An interpreter is a computing structure with one control point. One of the many available instructions is chosen at this control point based on the current state and inputs. The state is then processed by this instruction and the cycle begins again.

In general, a microprocessor specification can consist of many abstraction levels. Every level except the bottom specification (which is the structural specification) can be modeled as an interpreter. A hierarchical approach to specification and verification has been shown to significantly reduce the amount of effort required to complete the verification of a microprocessor [Win90b].

2.2 State.

At times it is convenient to treat state as an object of type $\mathbf{S}$, where $\mathbf{S}$ is uninterpreted. This allows us to treat state in an abstract manner, knowing nothing of its structure or content. Whether or not $\mathbf{S}$ is interpreted, we write $\mathbf{S} \leq \mathbf{S}'$ to indicate that $\mathbf{S}$ is an abstraction of $\mathbf{S}'$. The fact that $\mathbf{S}$ is an abstraction of $\mathbf{S}'$ implies that there exists a function, $\sigma : \mathbf{S}' \rightarrow \mathbf{S}$. The function σ is called the state abstraction function.

2.3 Time.

In general, different levels in the interpreter hierarchy have different views of time. A temporal abstraction function maps time at the abstract level to time at the con-

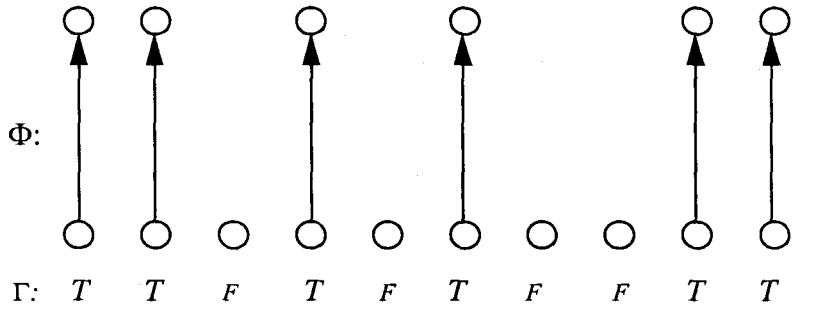

Figure 1. The function Φ, which maps time at one level to another, can be defined in terms of a predicate, Γ, which is true only when the mapping occurs.

crete level [Her88, Joy89, Mel88]. Figure 1 shows a temporal abstraction function Φ. The circles represent clock ticks. Notice that the number of clock ticks required at the concrete level to produce one clock tick at the abstract level is irregular.

The temporal projection, Φ, can be defined recursively on time. We define Φ in terms of a predicate, Γ, which is true whenever there is a valid abstraction from the concrete level to the abstract level. In a microprocessor specification, Γ is usually a predicate indicating when the lower level interpreter is at the beginning of its cycle—a condition that is easy to test. The function Φ is defined recursively so that $\Phi(\Gamma, 0)$ is the first time that Γ is true and $\Phi(\Gamma, (n+1))$ is the next time after time n when Γ is true. The resulting function is monotonically increasing. We use **N** to represent time. Thus, we define $\Phi : (\mathbf{N} \rightarrow \mathbf{B}) \times \mathbf{N} \rightarrow \mathbf{N}$ such that[1]

$$\forall\, n, m \cdot (n > m) \Rightarrow (\Phi(\Gamma, n) > \Phi(\Gamma, m))$$

We refer the interested reader to the references given above and [Win90a] for the details of the temporal abstraction function.

2.4 State Streams.

A state stream is a function from time to state, $\mathbf{N} \rightarrow \mathbf{S}$. An important part of our theory is the abstraction between state streams at different levels. State stream u is an abstraction of state stream s' (written $s \sqsubseteq s'$) if and only if

1. each member of the range of s is a state abstraction of some member of the range of s' and
2. there is a temporal mapping from time in s to time in s'.

There are two distinct kinds of abstraction going on: the first is a data abstraction and the second is a temporal abstraction. Using the state abstraction function, σ, and

1. **N** is the natural numbers and **B** is the booleans.

a temporal abstraction function, τ (defined in terms of Φ and Γ), we define stream abstraction as follows

$$s \subseteq s' \equiv \exists (\sigma : \mathbf{S}' \rightarrow \mathbf{S}) \cdot \exists (\tau : \mathbf{N} \rightarrow \mathbf{N}) \cdot \sigma \circ s' \circ \tau = s$$

where o denotes function composition.

2.5 Environments.

The environment represents the external world; it plays an important part in our theory. The environment is where interrupt requests originate, reset signals are generated, and so on. In our model, the environment is used only for input; output to the environment is assumed to be simply a function of the state and environment. At the abstract level, we treat the environment as an uninterpreted type. We know nothing about its structure or content. We denote it as **E**. Just as we defined σ, the state abstraction function, we define an environment abstraction function, ε, such that $\varepsilon : \mathbf{E}' \rightarrow \mathbf{E}$. When we provide an interpretation for ε, we represent the environment using n-tuples of booleans and bit–vectors. We perform the same kinds of abstraction on the environment as on states. Temporal abstraction is performed as it was for states. We define abstraction for environment streams in the same manner that we defined it for state streams. Thus, we write $e \subseteq e'$ when e is an stream abstraction of e' and define stream abstraction for environment streams as follows:

$$e \subseteq e' \equiv \exists (\varepsilon : \mathbf{E}' \rightarrow \mathbf{E}) \cdot \exists (\tau : \mathbf{N} \rightarrow \mathbf{N}) \cdot \varepsilon \circ e' \circ \tau = e$$

2.6 The Interpreter Specification.

The preceding parts of this section have given preliminary definitions for concepts important in the mathematical definition of interpreters. This section presents that definition. Interpreters are state transition systems. The difference between our model of interpreters and other models of state transition systems such as deterministic finite automata (*dfa*) is that our model accounts for state abstraction and aggregation. By state aggregation, we are referring specifically to *stores*. A store represents a collection of state that we deal with as a monolithic unit. In a *dfa* model, each location in memory is typically represented by a different piece of state which would be treated individually.

An interpreter, ***I***, is a predicate defined in terms of a 3–tuple, (***J***, ***K***, ***C***) where ***J***, ***K***, and ***C*** are defined as follows:

- Let **J** be the type of all functions with domain ($\mathbf{S} \times \mathbf{E}$) and codomain **S**. Not all functions in **J** are meaningful; the specifier's job is to choose meaningful functions. We use a subset of **J** to represent the instruction set; we call this set ***J***. The functions in ***J*** provide a denotational semantics for the instructions that they represent.
- In order to uniquely identify each instruction in ***J***, we associate it with a unique key. At the abstract level, we take keys from the uninterpreted domain **K**. At the concrete level, keys can have various representations. For example, in the top–

level specification of a microprocessor, the keys may represent opcodes. We must be able to choose instructions from J according to some predefined selection criteria. The selection is based on the current state and environment. We define K to be a function with domain ($\mathbf{S} \times \mathbf{E}$) and codomain $\mathbf{K}$.

We define C to be a choice function that has domain ($\mathbf{J} \times \mathbf{K}$) and codomain ($\mathbf{S} \times \mathbf{E} \rightarrow \mathbf{S}$). That is, C picks the state transition function from $\mathbf{J}$ that has a particular key in $\mathbf{K}$.

We define an interpreter, $I[s, e]$, as a predicate over the state stream, s, and the environment, e. The definition of I is given as

$$I[s, e] \equiv \forall\, t{:}\mathbf{N} \cdot s(t+1) = C(J, k\,t)\ (s\,t, e\,t)$$

where

$$k\ t = K(s\,t, e\,t)$$

The predicate constrains the state of the interpreter at time $t+1$ to be a function of the state and environment at time t. The function is determined by the instruction currently selected by K.

2.7 Interpreter Verification.

Our goal is to prove a correctness relation between the interpreters at different levels of a microprocessor abstraction. In particular, for two interpreters, I_m and I_l, we wish to show that

$$I_m[s_m, e_m] \Rightarrow I_l[s_l, e_l]$$

where s_m (e_m) is the state (environment) stream at level m, s_l (e_l) is the state (environment) stream at level l and $s_l \subseteq s_m$ ($e_l \subseteq e_m$). When this implication is true, I_l is an abstraction of I_m and I_m is said to *implement* I_l. The correctness theorem given above follows from the following lemma:

$$\forall j \in J \cdot I_m(s_m, e_m) \wedge j = C(J, k\,t) \Rightarrow \exists\, c \cdot (\sigma \circ s_m)\ (t+c) = j((\sigma \circ s_m)\ t, (\varepsilon \circ e_m)\ t)$$

This lemma, which we call the instruction correctness lemma, states that every instruction follows from the concrete interpreter, I_m. Specifically, it says that for every instruction, j in J, if j is selected, then applying j to the current abstract state and environment, $(\sigma \circ s_m)\ t$ and $(\varepsilon \circ e_m)\ t$, yields the same abstract state that results from letting the concrete interpreter I_m run for c cycles. The instruction correctness lemma suggests a case analysis on the instruction set. In addition, the instruction correctness lemma ignores temporal abstraction, stating only that there exists a time in the future when the states correspond. Thus, the proof obligation on the user of the generic interpreter theory has little to do with the temporal abstraction reasoning necessary to verify a microprocessor. That is all contained in the abstract theory. This lemma plays an important role in the work which we describe next.

3 A Formal Model of Interpreters.

This section presents the our generic interpreter theory for the HOL verification system. The basic structure is the same as presented in the last section. In addition to the correctness result, however, we prove several other important theories about interpreters including an induction theorem and theorem about hierarchical composition of interpreters.

3.1 Abstract Theories.

A abstract theory is parameterized so that some of the types and constants defined in the theory are undefined inside the theory except for their syntax and a loose algebraic specification of their semantics. Abstract theories are useful because they provide proofs about abstract structures which can be used to reason about specific instances of the structure.An abstract theory consists of three parts:

1. An *abstract representation* of the uninterpreted constants and types in the theory.
2. A set of *theory obligations* defining relationships between members of the abstract representation. Inside the theory, the obligations represent axiomatic knowledge concerning the abstract representation. Outside the theory, the obligations represent the criteria that a concrete representation must meet if it is to be used to instantiate the abstract theory.
3. A collection of *abstract theorems*. The theorems are generally based on the theory obligations and can stand alone only after the theory obligations have met.

To instantiate an abstract theory, the concrete representation must meet the syntactic requirements of the abstract representation as well as the semantic requirements of the theory obligations. If the syntactic and semantic requirements are met, then the instantiation provides a collection of concrete theorems about the new representation.

There are several specification and verification systems that support abstract theories. Some, such as OBJ [Gog88] and EHDM [SRI88], offer explicit support. HOL, the verification environment used for the research reported here, does not explicitly support abstract theories; however, HOL's metalanguage, ML, combined with higher–order logic, provides a framework for concrete abstract theories in a manner that does not degrade the trustworthiness of the theorem prover. See [Win92] for details about using abstract theories in HOL

3.2 The Abstract Representation

We specify the abstract representation by defining a list of abstract objects and operations. Table 2 shows the operations and their types.

We must emphasize that the representation *is* abstract and, therefore, the objects and operations have no definitions. The descriptions that follow are what we *intend* for

Table 1 The abstract functions and their types for the generic interpreter model.

Operation	*Signature*
instructions	`:*key->(*state->*env->*state)`
select	`:*state->*env->*key`
output	`:*key->(*state->*env->*out)`
substate	`:*state'->*state`
subenv	`:*env'->*env`
subout	`:*out'->*out`
implementation	`:(time'->*state')->(time'->*env')->bool`
sync	`:*state'->*env'->bool`

the representation to mean. The representation is purely syntactic, however.The following abstract types are used in the representation.

- `:*state` represents the state and corresponds to **S** from the last section.
- `:*env` represents the environment and corresponds to ***E*** from the last section.
- `:*out` represents the outputs. In the model in the last section, outputs were assumed to be a function of the current state and environment. In the formal model we will represent this explicitly.
- `:*key` is type containing all of the keys and corresponds to ***K*** from the last section.

The abstract representation can be broken into three parts. The first contains those operations concerned with the interpreter.

- `instructions` is the instruction set. The set is represented by a function from a key to a state transition function and corresponds to ***J*** from the last section.
- `select` picks a key based on the present state and environment and corresponds to ***K*** from the last section.
- `output` is a set of output functions. The set is represented by a function from a key to a function that produces output for a given state and environment.

The second part contains the abstraction functions:

- `substate` is the state abstraction function for the interpreter and corresponds to σ from the last section.
- `subenv` is the environment abstraction and corresponds to ε from the last section.
- `subout` is the output abstraction.

Because we want to prove correctness results about the interpreter, we must have an implementation. The third part of the abstract representation contains three functions which provide the necessary abstract definitions for the implementation.

- `implementation` is the abstract implementation. We could have chosen to make this function more concrete, but doing so would require that every implementation have some pre-chosen structure. Thus, we say nothing about it except to define its type.
- `sync` is the synchronization predicate for the temporal abstraction and corresponds to Γ from the last section.

The components of the last part of the abstract representation correspond to concrete interpreter from in level below the abstract interpreter we are defining.

3.3 The Theory Obligations

Proving that the implementation implies the interpreter definition is typically done by case analysis on the instructions; we show that when the conditions for an instruction's selection are right, the instruction is implied by the implementation. We call this the instruction correctness lemma.

The predicate `INSTRUCTION_CORRECT` expresses the conditions that we require in the instruction correctness lemma:[1]

```
⊢def INST_CORRECT gi s' e' p' k =
       (implementation gi s' e' p') ==>
       (!t:time'.
       let s  t = substate gi (s' t) in
       let e  t = subenv gi (e' t) in
       let f  t = sync gi (s' t) (e' t) in (
       (select gi (s t) (e t) = k) /\
       (f  t) ==>
       ? c. Next f (t,t+c) /\
        (instructions gi k (s t) (e t) = (s(t + c)))))
```

`INSTRUCTION_CORRECT` operates on a single key, `k`. This theory obligation requires that the implementation imply that for every time, `t`, if `k` is the key returned by `select` and the synchronization predicate is true, then there is time `c` cycles in the future such that applying the instruction selected by `k` to the current state yields the same state change that the implementation does in `c` cycles.

`INSTRUCTION_CORRECT` is a good example of the kind of information that is captured in the generic model. Previous microprocessor verifications created this lemma, or one similar to it, in a largely *ad hoc* manner.

---

1. The HOL code in the remainder of the paper is shown using the HOL convention of representing universal quantification, existential quantification, lambda quantification, implication, conjunction, disjunction, and negation by the symbols !, ?, \, ==>, /\ \/, and ~ respectively. The form "e1 => e2 | e3" represents "if e1 then e2 else e3."

Because our model has outputs as well as inputs (the environment), we must also assume something about the output in order to establish correctness. The predicate OUTPUT_CORRECT expresses the conditions that we require in the output correctness lemma:

```
⊢def OUTPUT_CORRECT gi s' e' p' k =
       (implementation gi s' e' p') ==>
       (!t:time'.
       let s t = substate gi (s' t) in
       let e t = subenv gi (e' t) in
       let p t = subout gi (p' t) in
       let f t = sync gi (s' t) (e' t) in (
       (select gi (s t) (e t) = k) /\
       (f t) ==>
       (p t = (output gi k) (s t) (e t))))
```

OUTPUT_CORRECT is similar to INSTUCTION_CORRECT. The major difference is that output is assumed to happen instantaneously and thus there are no temporal considerations.

Using INSTRUCTION_CORRECT and OUTPUT_CORRECT we can define the theory obligation for our model. The theory obligations are given as a predicate on an abstract representation gi:

```
⊢def GI gi =
       (!s' e' p' k. INST_CORRECT gi s' e' p' k)  /\
       (!s' e' p' k. OUTPUT_CORRECT gi s' e' p' k)
```

The predicate says that every instruction in the instruction set satisfies the predicate INSTRUCTION_CORRECT and every output function satisfies the conditions set forth in OUTPUT_CORRECT.

3.4 Abstract Theorems

Using the abstract representation and the theory obligations, many useful theorem pertaining to interpreters can be established on the generic structure.

Defining the Interpreter. One of the important parts of the collection of abstract theorems is the definition of a generic interpreter. The definition is based on functions from the abstract representation.

```
⊢def INTERP gi s e p =
       !t:time.
       let k = (select gi (s t) (e t)) in
       (s(t+1) = (instructions gi k) (s t) (e t)) /\
       (p t = (output gi k) (s t) (e t))
```

The specification of an interpreter is a predicate relating the contents of the state stream at time t+1 to the contents of the state stream at time t. The relationship is defined using the functions from the abstract representation. The definition also uses the currently selected output function to denote the current output.

Induction on Interpreters. The definition of the interpreter sets up a relation between the state at t and t+1. Sometimes it is useful to have a more explicit statement regarding induction. The following theorem, which follows from the definition of the interpreter given in Section 3.4.1, defines induction on an interpreter:

```
⊢ ! Q. INTERP gi s e p ==>
     (Q (s 0) /\
      !t. let inst = (instructions gi
                        (select gi (s t) (e t))) in (
          Q (s t) ==> Q (inst (s t) (e t)))) ==>
      !t. Q (s t)
```

The theorem states that for any arbitrary predicate on states, `Q`, if `Q` is true of the state at time 0 and when `Q` is true of the state at time t, it follows that its also true of the state returned by the current instruction, then `Q` is true of every state.

We note that even though this theorem looks fairly simple, and indeed is quite easy to show in the generic theory, the theorem will eventually be instantiated with the entire denotational description of the semantics of a particular instruction set and will be quite involved. The same admonition holds for each of the theorems and definitions presented in this section.

The Implementation is Live. Using the theory obligations, we can prove that the implementation is live. By *live* we mean that if the implementation starts at the beginning of its cycle, then their is a time in the future when the implementation will be at the beginning of its cycle again. That is, we show that the device will not go into an infinite loop.

```
⊢  implementation gi s' e' p' ==>
   (!t.(sync gi (s' t) (e' t)) ==>
       (?n. Next(\t. sync gi (s' t)(e' t))(t,t + n)))
```

`Next P (t1, t2)` says that `t2` is the next time after `t1` when `P` is true.

The Correctness Statement. The correctness result can be proven from the definition of the interpreter and the theory obligations:

```
⊢ let  s  t = substate gi (s' t) and
       e  t = subenv gi (e' t) and
       p  t = subout gi (p' t) and
       g  t = sync gi (s' t) (e' t) in
   let abs = Temp_Abs f in
   (implementation gi s' e' p') /\
   (?t. f t) ==>
   (INTERP gi) (s o abs) (e o abs) (p o abs)
```

In the correctness statement, `s'`, `e'`, and `p'` are the state, environment, and output streams in the implementation. The function `abs` is defined in terms of a general purpose temporal abstraction functions, `Temp_ABS`, corresponding to Φ and a predicate, `g`, corresponding to Γ. The terms `(s o abs)`, `(e o abs)`, and `(e o abs)` are the state, environment, and output streams for the interpreter defined in the model. They are data and temporal abstractions of `s'`, `e'`, and `p'`. The correct-

ness statement says that if the implementation is valid on its state, environment, and output streams and there is a time when the concrete clock is at the beginning of its cycle, then the interpreter is valid on its state and environment streams.

Vertical Composition. In [Win90b], we show that hierarchical decomposition makes the verification of large microprocessors practical. To support this decomposition, the generic interpreter model contains theorems about vertically composing generic interpreters.

More generally, we can say that any two generic interpreters can be composed to form another generic interpreter as long as the implementation of one is the interpreter of the other. We define a composition operator as follows:

```
⊢def GI_VERT_COMP gi1 gi2 =
        GI ( (instructions gi2)
             (select gi2)
             (output gi2)
             ((substate gi2) o (substate gi1))
             ((subenv gi2) o (subenv gi1))
             ((subout gi2) o (subout gi1))
             (implementation gi1)
             (\s e .
                 (sync gi1 s e) /\
                 (sync gi2 (substate gi1 s)
                           (subenv gi1 e)))
```

The resulting structure composes the data abstractions using function composition and requires that the synchronization predicates at *both* levels be true.

We can prove that the structure resulting from such a composition is a generic interpreter (i.e. it has all the properties of a generic interpreter) under a single restriction:

```
⊢ (INTERP gi1 = implementation gi2) ==>
        IS_GI(GI_VERT_COMP gi1 gi2)
```

Provided that the interpreter defined by the first is the implementation of the second, the resulting structure *is* a generic interpreter. This theorem is more generally useful since we can prove the theory obligations of each level of the hierarchy separately, show that the composition of these separate results is a generic interpreter using this theorem, and then use the result to instantiate the correctness theorem from section 3.4.4 to show that the bottom–most member of the hierarchy implies the top–most member.

A further result shows that the composition operator is associative and, therefore, the order of the composition is unimportant:

```
⊢ GI_VERT_COMP gi1 (GI_VERT_COMP gi2 gi3) =
  GI_VERT_COMP (GI_VERT_COMP gi1 gi2) gi3
```

The generic interpreter theory contains the structure for the entire proof, freeing the user from worrying about the data and temporal abstractions that result from the composition. The theorems about vertical composition are a good examples of the

utility of abstract theories in hardware verification. The theorems are tedious to prove in specific cases and were they not contained in the abstract theory, they would have to be proven numerous times in the course of a single microprocessor verification.

4 Conclusion

This paper has described the generic interpreter model. The theory isolates the temporal and data abstractions of the proof inside the abstract theory. The theory also contains several important theorems about the abstract representation. These theorems are true of every instantiation of the abstract representation that meets the theory obligations. The theory has important benefits:

- The generic model structures the proof by stating explicitly which definitions must be made (one for each of the members of the abstract representation) and which lemmas need to be proven about these definitions (namely, the theory obligation). This is a substantial improvement over previous microprocessor verifications where these decisions were made on an *ad hoc* basis.
- The generic model insulates users of the model from complex proofs about the data and temporal abstractions. These proofs are done once and then made available to the user by instantiation.
- The use of a generic interpreter model for specifying and verifying microprocessors provides a methodological approach. Making specification and verification methodological is an important step in turning what has been primarily a research activity into an engineering activity.

We have used the generic interpreter theory to verify a microprocessor, *AVM–1*, with a modern load–store architecture [Win90a]. Other efforts to use the generic interpreter theory are underway. We believe that our methodology makes microprocessor verification accessible by non–experts. We are testing our belief by using the generic interpreter theory to introduce microprocessor verification to graduate students with no previous verification experience [Coe92].

Based on our experience with *AVM–1*, we are confident that the generic interpreter theory makes microprocessor specification and verification significantly easier because of the structure that it entails and the theorem reuse that it enables.

5 References

[Aro90] Tejkumar Arora. The formal verification of the VIPER microprocessor: EBM to microcode level. Master's thesis, University of California, Davis, 1990.

[Coe92] Michael L. Coe and Phillip J. Windley. Using the Generic Interpreter Theory to Verify Microprocessors: A Tutorial. Technical Report LAL–92–10, University of Idaho, Department of Computer Science, Laboratory for Applied Logic. December, 1992.

[Coh88] Avra Cohn. Correctness properties of the VIPER block model: The second level. Technical Report 134, University of Cambridge Computer Laboratory, May 1988.

[SRI88] SRI International Computer Science Laboratory. *EHDM Specification and Verification System: User's Guide*, Version 4.1, 1988.

[Gor88] Michael J.C. Gordon. HOL: A proof generating system for higher-order logic. In G. Birtwhistle and P.A Subrahmanyam, editors, *VLSI Specification,Verification, and Synthesis*. Kluwer Academic Press, 1988.

[Gog88] J. Goguen and T. Winkler. Introducing OBJ3. Technical Report SRI-CSL-88-9, SRI International, August 1988.

[Her88] John Herbert. Temporal abstraction of digital designs. In G.J. Milne, editor, *The Fusion of Hardware Design and Verification, Proceedings of the IFIP WG 10.2 International Working Conference, Glasgow, Scotland*. North–Holland, 1988.

[Hun87] Warren A. Hunt. The mechanical verification of a microprocessor design. In D. Borrione, editor, *From HDL Descriptions to Guaranteed Correct Circuit Designs*. Elsevier Scientific Publishers, 1987.

[Hun89] Warren A. Hunt. Microprocessor design verification. *Journal of Automated Reasoning*, Vol 5, pages 429–460, 1989.

[Joy89] Jeffrey J. Joyce. *Multi–Level Ver*ification of Microprocessor–Based Systems. PhD thesis, Cambridge University, December 1989.

[Low89] Paul Loewenstein. Reasoning about state machines in higher–order logic. In M. Leeser and G. Brown, editors, *Workshop on Hardware Specification, Verification, and Synthesis: Mathematical Aspects*, Lecture Notes in Computer Science. Springer-Verlag, 1989.

[Mel88] Thomas Melham. Abstraction mechanisms for hardware verification. In G. Birtwhistle and P. A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*. Kluwer Academic Publishers, 1988.

[Win90a] Phillip J. Windley. *The Formal Verification of Generic Interpreters*. PhD thesis, University of California, Davis, Division of Computer Science, June 1990.

[Win90b] Phillip J. Windley. A hierarchical methodology for the verification of microprogrammed microprocessors. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 1990.

[Win91] Phillip J. Windley. The formal specification of a high–speed CMOS correlator. In *Proceedings of the Third Annual IEEE/NASA Symposium on VLSI Design*, October 1991.

[Win92] Phillip J. Windley. Abstract Theories in HOL. In *Proceedings of the 1992 International Conference on the HOL theorem Prover and its Application*, October 1992.