An Interpreter Interface Language: From Its Formal Embedding in Higher-Order Logic to Its Role in Better Simulation Practices

David A. Fura[1,2] and Arun K. Somani[2,3]

[1] Flight Critical Information Processing, Boeing Defense & Space Group, Seattle, WA, 98124, USA
[2] Electrical Engineering Department, University of Washington, Seattle, WA, 98195, USA
[3] Computer Science & Engineering Department, University of Washington, Seattle, WA, 98195, USA

Abstract. Interpreters are the model of choice for much of the hardware design-verification community. In part, by restricting behavior to a single level of abstraction, interpreters provide a highly desirable conciseness of expression. However, this single-level extent necessarily requires some other means to 'bridge the gap' between interpreters in a multi-level hierarchy; i.e., to define the abstraction between the levels. This paper presents an interpreter interface language that we have found to be well suited to the task of modeling interpreter abstraction. We describe our formal embedding of this language within the HOL theorem proving system, and we overview some of the proof support provided by our implementation. Finally, we describe a fundamental role of interface languages in the practical insertion of formal methods concepts into mainstream simulation environments.

1 Introduction

Finite-state machines (FSMs) are the modeling workhorses for digital system design and verification. Among their advantages, FSMs are executable, composable, concise, and widely understood within the hardware design and verification communities. An *interpreter* is a form of FSM where the next-state and output behavior is partitioned into a set of instructions. Interpreter modeling techniques can be readily applied to virtually any digital computing system.

Interpreters provide simple and easily understood specifications, in part, because they represent behavior at a *single level of abstraction*. However, this single-level extent necessarily requires some other means to 'bridge the gap' between interpreters in a multi-level hierarchy. A formalism is needed that can relate the signals of an abstract interpreter model to those of a concrete implementation model; i.e., to define the *abstraction* between the two levels. We call such a formalism an *interface language*.

In our view, interpreter interface languages have a key role in bridging a second important gap: that existing between current hardware design practice and the formal approaches being pursued by the hardware research community. Almost all practical hardware modeling and verification today is done using simulation languages such as VHDL [IEE88]. Simulation typically involves a highly labor-intensive interaction between the hardware designers and a detailed model of the implementation. Simulation is increasingly perceived to be overly expensive and ever less effective in the face of continuing increases in system complexity.

Making formal methods acceptable to the mainstream design community will certainly require advances in verification tools, but we should not overlook the key role of *modeling formalisms* in this process. Many designers dislike working with raw predicate logic and find existing temporal logics (e.g., CTL [McM93]) too austere. On the other hand, many designers also find it unacceptable that they lack effective techniques both to specify their designs at high levels of abstraction and to express, in a rigorous way,

This research was partially funded by NASA-Langley Research Center under contract NAS1-18586, Task 10. The NASA technical monitor was Sally Johnson.

abstraction relationships. We believe that a strong focus on interface language development is an important part of a practical approach to transferring formal methods into working design environments.

In this paper we describe a prototype language specifically targeted to problems in modeling interpreter abstraction. This language has a number of temporal-logic style constructs that permit concise representation of complicated relationships expressed at points of time or over intervals of time. An underlying design guideline for this language has been to strive for a familiar and natural syntax to promote the widest possible acceptance within the hardware design community. We have been encouraged so far by a favorable reaction from our local designers.

We have embedded this interface language within the HOL theorem proving system [Gor93], and have applied it to problems in the formal modeling and verification of a fault-tolerant hardware system (e.g., see [Fur94a]). In addition, we have also developed a new abstraction language targeted specifically to VHDL simulation environments. This language uses many of the constructs described in this paper, but is embedded within a VHDL-style syntax. We do not describe this language here, but only explain how it serves the automated construction of simulation testbench models for an advanced simulation environment. This environment is described in [Fur94b] and is the subject of a future paper.

Table 1 captures the important syntactic structures of our language and provides an informal interpretation of their meanings.[1] In this paper we describe the implementation of these structures in some detail and summarize some of their established proof support. This is done in Sections 2 and 3, respectively. Section 4 introduces the role of this language in simulation environments. Section 5 presents our conclusions.

Table 1: Overview of Major Interface Language Constructs.

Language Syntax	Informal Interpretation
f At t	the value of signal f at time t
f Since t	the value of signal f beginning at time t
f During (t1, t2)	f is true at every time in the closed interval [t1, t2]
f Within (t1, t2)	f is true for at least one time within the closed interval [t1, t2]
(f While g) Since t1	beginning at time t1, f is true while g is true
(Henceforth f) Since t1	beginning at time t1, f is henceforth true
(f Until g) Since t1	beginning at time t1, f is true until g is true (strong until)
(Eventually f) Since t1	beginning at time t1, f is eventually true
(f For n ’thCountSince t1) At t2	f is true at time t2 for the n’th count since time t1
(f AtLeast n CountsSince t1) At t2	f is true at least n counts within the closed interval [t1, t2]
(f FewerThan n CountsSince t1) At t2	f is true fewer than n counts within the closed interval [t1, t2]
(f Exactly n CountsSince t1) At t2	f is true exactly n counts within the closed interval [t1, t2]
TimeWhen (f For n ’thCountSince t1)	the time when f is true for the n’th count since time t1

[1] In this paper, all *counts* start at zero.

2 Description of Major Language Constructs

In this section we describe the language constructs contained in Table 1. They are covered in the five subsections below in roughly the same order and grouping used in the table. Before doing this we point out that much of the syntax of our language is actually not covered in this paper. These forms are mainly concerned with generating compound signals out of simpler signals. For example, one that is used later in this section creates the negation of a boolean signal. It is defined as: $\vdash_{def}$ **Not f = λt. ¬ f t.** It and the other signal-generation operators create the signals, **f**, that are used in the table and throughout this section.

2.1 The Point Primitives: At and Since

The operators **At** and **Since** act on a single point of time, returning the value of a signal there. Their HOL definitions are shown next. One could argue that the importance of these operators is low, however, we like them because they help maintain a consistent style of syntax. Even though these two operators describe the same behavior, Table 1 reveals their unique roles in promoting comprehensible syntax.

Definitions: At and Since

$\vdash_{def}$ **f At t = f t**
$\vdash_{def}$ **f Since t = f t**

2.2 The Interval Primitives: During and Within

During and **Within** are the two most fundamental interval predicates that we use. **During** characterizes intervals where a boolean signal is true at every time in the interval (if the interval exists). We define **Within** in terms of **During** as shown below, with the theorem **WITHIN_THM** providing a somewhat clearer description. This theorem says that a time does exist within the interval for which the signal is true; in contrast to **During**, it requires the existence of a non-empty interval.

Definitions for interval primitives: During and Within

$\vdash_{def}$ **f During (t1, t2) = ∀t. t1 ≤ t ∧ t ≤ t2 ⊃ f t**
$\vdash_{def}$ **f Within (t1, t2) = ¬ (Not f) During (t1, t2)**

Theorem: WITHIN_THM

$\vdash$ **f Within (t1, t2) = ∃t. t1 ≤ t ∧ t ≤ t2 ∧ f t**

Our use of **During** directly follows a similar version used in the HCL language of Joyce and Seger [Joy93]. In fact, this operator has an extensive history now, having been used (at least) in the work of Hanna & Daeche [Han86], Herbert [Her88], and Melham [Mel90]. It is also similar to the (interval) **always** operator of Moszkowski's interval temporal logic [Mos85]. Our definition of **Within** is a generalized form of the **between** operator used by Melham.

2.3 The Primitive Temporal Operators: While, Henceforth, Until, and Eventually

The operators of this section are defined quite simply in terms of the operators **During** and **Within**. **While** defines a signal carrying true precisely when its left argument's obligation to be continuously true in the future is released only by a false value for the right argument, an event that may or may not occur. **Henceforth** describes a signal carrying true precisely when its argument is true for all future times. **Until** defines a signal carrying true precisely when its left argument is true until its right argument is true, and the right argument is eventually true. **Eventually** describes a signal carrying true precisely when its argument is true for some future time.

> **Definitions: While, Henceforth, Until, and Eventually**
>
> $\vdash_{def}$ **f While g = λt. ∀t2. g During (t, t2) ⊃ f During (t, t2)**
> $\vdash_{def}$ **Henceforth f = f While True**
> $\vdash_{def}$ **f Until g = λt. ∃t2. g Within (t, t2) ∧ f During (t, t2–1)**
> $\vdash_{def}$ **Eventually f = True Until f**
>
> **Theorems: HENCEFORTH_THM and EVENTUALLY_THM**
>
> $\vdash$ **(Henceforth f) Since t1 = ∀t2. f During (t1, t2)**
> $\vdash$ **(Eventually f) Since t1 = ∃t2. f Within (t1, t2)**

The operators in this section come closest to the propositional temporal logic primitives in widespread use today: henceforth, eventually, next, and until (e.g., [Man92]). Others have embedded operators in HOL more faithful to the original definitions, including Jasuja [Jas90] and Joyce [Joy90].

2.4 The Event-Count Temporal Operator: 'thCountSince (plus TimeWhen)

In this section, we present a new, more general, signal-based version of Melham's **Istimeof** predicate [Mel90]. The underlying foundation for the syntax in this section is the predicate **NthTimeAt**, which is itself based on the **NextAt** and **From** operators. The description in this section proceeds 'bottom-up.'

The following HOL segment shows the definitions for **NextAt** and **From**. The form **f NextAt t2** is a signal carrying true precisely when **f** is next true at **t2**. The combined predicate **From t1 (f NextAt t2)** says that, beginning at time **t1**, **f** is next true at **t2**.

> **Definitions: NextAt and From**
>
> $\vdash_{def}$ **f NextAt t2 = λt. f Within (t, t2) ∧ (1 ≤ t2 ⊃ ¬ f Within (t, t2–1))**
> $\vdash_{def}$ **From t f = f t**
>
> **Theorem: FROM_NEXT_AT_THM**
>
> $\vdash$ **From t1 (f NextAt t2) = t1 ≤ t2 ∧ f t2 ∧ (1 ≤ t2 ⊃ (Not f) During (t1, t2–1))**

This predicate represents a standard type of behavior similar to previous work elsewhere. It is the basis for the primitive event-count operator **NthTimeAt** described next. The predicate **NthTimeAt n f t1 t2** says that **f** is true at time **t2** for the **n** 'th count since time **t1**. The main difference between this and Melham's **Istimeof** is the addition of the specified lower time bound **t1**.

> **Definition for event-count primitive: NthTimeAt**
>
> $\vdash_{def}$ **NthTimeAt 0 f t1 t2 = From t1 (f NextAt t2) ∧**
> **NthTimeAt (SUC n) f t1 t2 = ∃t. NthTimeAt n f t1 t ∧ From (t+1) (f NextAt t2)**

The definitions used to produce the syntax shown in Table 1 operate in multiple steps. As shown in the following HOL segment, the infix operator **'thCountSince** packages its two inputs (**n** and **t1**) into a 2-tuple. The infix operator **For** takes its two inputs **f** and **(n, t1)** and produces a signal, demonstrated by the theorem **FOR_THM**.

Definitions: 'thCountSince and For

$\vdash_{def}$ **n 'thCountSince t1 = (n , t1)**

$\vdash_{def}$ **f For (n , t1) = λ t. NthTimeAt n f t1 t**

Theorem: FOR_THM

|- **f For n 'thCountSince t1 = λ t. NthTimeAt n f t1 t**

This fact about **For** is a key one and was carefully designed into the above definitions. When combined with the definition for **At** and the following definition for **TimeWhen**, the signal constructed by **For** supports the intended semantics of Table 1.

Definition: TimeWhen $\vdash_{def}$ **TimeWhen f = ε t. f t**

Two important concepts of temporal abstraction are the 'instruction boundary predicate' (or **NthTimeAt n f t1 t2** here), and the concrete time mapped by the temporal abstraction function (or ε **t. NthTimeAt n f t1 t2** here). As shown in the following HOL segment, the signal produced by **For**, when combined with the definitions of **At** and **TimeWhen**, produce precisely these values, and do so using the syntax of Table 1.

Theorems: AT_TIME_THM and AT_TIME_WHEN_THM

|- **(f For n 'thCountSince t1) At t2 = NthTimeAt n f t1 t2**

|- **TimeWhen (f For n 'thCountSince t1) = ε t. NthTimeAt n f t1 t**

2.5 The Quantitative Temporal Operators : AtLeast, FewerThan, and Exactly

The predicate **NthTimeWithin** plays a similar role in this section as **NthTimeAt** of the last section. Its definition is shown next. The predicate **NthTimeWithin n f t1 t2** says that the **n**'th count of **f** occurs within the closed interval [**t1**, **t2**]. Note that this definition does not say **f** occurs *exactly* **n** counts within the interval – it may in fact occur more than this.

Definition: NthTimeWithin

$\vdash_{def}$ **NthTimeWithin n f t1 t2 = (λ t. NthTimeAt n f t1 t) Within (t1, t2)**

Rather than show the actual definitions for the various operators that create the syntax of Table 1, we instead present the theorems that more concisely characterize the concepts of 'at least,' 'fewer than,' and 'exactly' in terms of **NthTimeWithin**. These results are expected. For example, **f** occurs fewer than **n** counts within [**t1**, **t2**] precisely when it is not the case that the **n**'th count of **f** occurs within [**t1**, **t2**]. Likewise, **f** occurs exactly **n** counts precisely when it occurs at least **n** counts, but fewer than **n+1** counts.

Theorems: AT_LEAST_THM, FEWER_THAN_THM, and EXACTLY_THM

|- **f AtLeast n CountsSince t1 = λ t. NthTimeWithin n f t1 t**

|- **f FewerThan n CountsSince t1 = λ t. ¬ NthTimeWithin n f t1 t**

|- **f Exactly n CountsSince t1 = λ t. NthTimeWithin n f t1 t ∧ ¬ NthTimeWithin (n+1) f t1 t**

To our knowledge, the definitions presented in this section are unique to this work. We have used these three constructs quite extensively in hardware and fault-tolerant system modeling; in hardware modeling they provide a concise description of counters, for example. They demonstrate again the tremendous flexibility provided by the HOL system to define meaningful language constructs.

3 Theorem Proving Support

Our interface language theory currently contains over 80 proved theorems available to help speed up language-specific theorem proving. Virtually all of these were proven to meet some specific need during hardware circuit verifications. In this section we briefly overview some of the major classes of theorems; those at the end of this section are probably of more interest to most readers.

A large group of theorems allow us to transfer interval properties to sub-intervals and super-intervals. These are much like the theorems presented in [Her88], for example. The two shown here are representative of this class; these two allow us to conclude facts about right sub-intervals.

Theorems: RIGHT_SUB_DURING and RIGHT_SUB_WITHIN

|- f During (t1, t3) ∧ t1 ≤ t2 ⊃ f During (t2, t3)
|- f Within (t1, t3) ∧ ¬ f Within (t1, t2) ⊃ f Within (t2+1, t3)

Some of the most useful theorems allow us to convert wishy-washy statements like ‘within’ and ‘eventually’ into more substantial statements like ‘next at.’ The first theorem below is similar to one proven by Windley in an earlier version of this theory.

Theorems: FIRST_WITHIN and EVENTUALLY_IMP_NEXT_AT_EXISTS

|- f Within (t1, t2) ⊃ ∃ t. From t1 (f NextAt t) ∧ t ≤ t2
|- (Eventually f) Since t1 ⊃ ∃ t2. From t1 (f NextAt t2)

The next two theorems establish certain monotonicity results. The first is representative of a large group of theorems for **NthTimeAt**, similar to those of Graham [Gra92] and others. The second theorem is a comparable result for the quantitative predicate **AtLeast**. Note that we have raised our theorem-proving support to the level of the language syntax here, rather than keeping it at the level of **NthTimeWithin**. This is a verbosity issue – the **’thCountSince** syntax is somewhat more verbose than **NthTimeAt**; but this is not the case for the quantitative syntax and **NthTimeWithin**. Several theorems have been proven establishing the expected quantitative relationships between **AtLeast**, **FewerThan**, and **Exactly**.

Theorems: NTH_TIMES_MONO and AT_LEAST_COUNTS_MONO

|- NthTimeAt n1 f t0 t1 ∧ NthTimeAt n2 f t0 t2 ∧ n1 < n2 ⊃ t1 < t2
|- (f AtLeast n CountsSince t1) At t2 ∧ m ≤ n ⊃ (f AtLeast m CountsSince t1) At t2

After a history of bouts with the Hilbert choice operator that have not ended well, we took great pleasure in proving the following theorem, which says that **f** occurs fewer than **n** counts before the time at which **f** is true for the **n**’th count. Recall that **TimeWhen** introduces the choice operator into its argument.

Theorem: FEWER_THAN_BEFORE_TIME_WHEN

|- let t = TimeWhen (f For n ’thCountSince t1) in
(1 ≤ t1 ⊃ (f FewerThan n CountsSince t1) At (t–1))

We have found this theorem to be useful in hardware verifications. Ultimately, when verifying circuits it becomes necessary to link up the temporal variables (the end points of intervals for example) to the events that 'define' them. For example, **t2** might be the time of the first occurrence of some event **f**, defined using **TimeWhen**. To link up this time and event in a circuit proof we would have a statement of the form:

f FewerThan 0 CountsWithin (t1, t2–1) $\supset$ (***behavior expressed using*** **t2**)

With the above theorem we can directly establish the antecedent here and obtain the desired expression for the circuit behavior on the right hand side.

4 The Role of Interface Languages in Simulation Environments

Current hardware simulation methods are generally labor intensive and time consuming. While requiring large amounts of computer processing time and hands-on engineering effort, they are becoming less effective at ensuring the correctness of increasingly-complex hardware designs. Two basic simulation approaches see widespread use today.

The first method is 'test vector-based.' It requires the explicit generation of simulation test inputs and the application of these inputs to the design model. This method is especially labor intensive and usually suffers either from the lack of an explicit specification model or else an absence of a rigorous mapping between the specification and design. In either case, the selection of simulation inputs and the certification of simulation outputs are both time consuming and error prone. Prospects for significant automation are poor, given the lack of an adequate specification model.

The second method is 'testbench-based.' Here, a detailed testbench model is designed alongside the design model. The testbench supplies the test inputs to the design model so that these do not need to be generated by hand. This approach improves the efficiency of simulation-time testing since the testbench can convert abstract instructions, provided by the user, into the detailed input sequences required by the design. Similarly, it can package up detailed design output sequences into abstract entities for review by the user. The problem with this approach is that the construction of the testbench can be as difficult as the design itself.

Figure 1 shows a new simulation environment that we are developing to improve the rigor and automation of test-based verification. This approach augments the design model and its testbench with models for the specification and its testbench. In addition to the normal simulation-time advantages of design testbenches, this new approach supports the automatic certification of design outputs shown at the right hand

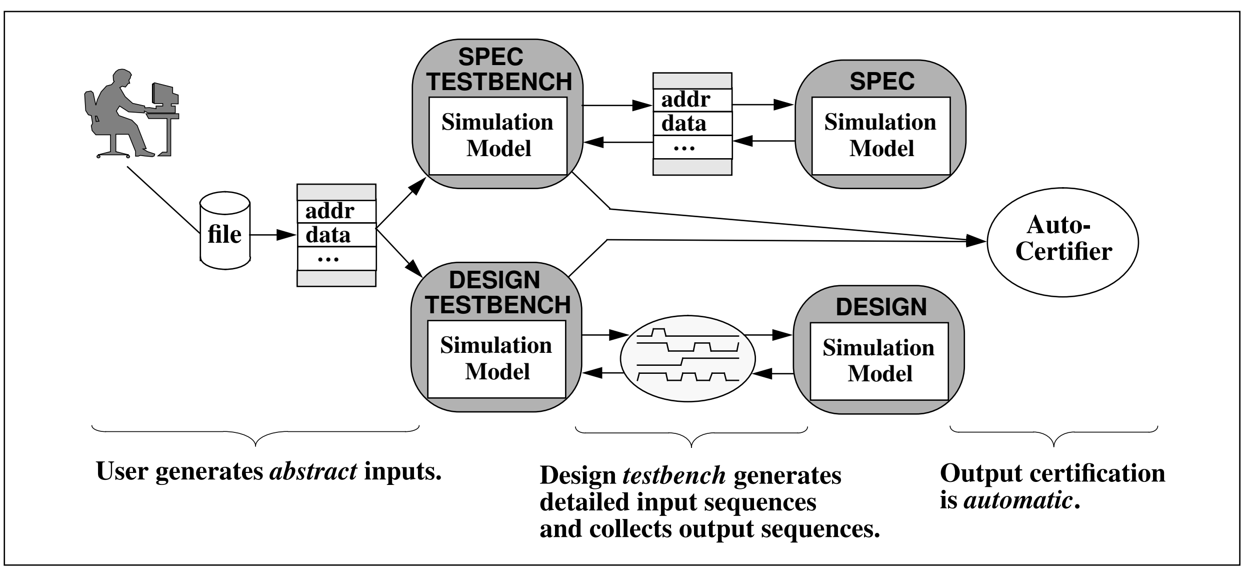

Figure 1: Simulation Environment with Enhanced Rigor and Automation.

side of the figure. It is clearly the presence of the specification model, acting on the same abstract packets as the design, that makes this possible.

To make this simulation environment practical requires efficient procedures for generating the associated models. Of the four models shown in Figure 1, we note that three are new, with respect to the vector-based simulation approach above. Of these, only the design testbench is expected to be a complex model, roughly on the order of the complexity of the design itself. The other two are expressed at the abstract level of the specification and are much simpler.

In Figure 1 we observe the design testbench at the crossroads linking signal streams expressed at two different levels of abstraction. To the left of the model are abstract packets, while to the right are detailed sequences that represent low-level behavior. In converting packets into sequences for design-model input, and then reversing the process on design-model output, the interface model is clearly implementing the abstraction linking the design to the specification.

It is clear that the manual construction of design testbenches implementing this abstraction can be very difficult; again, depending on the design, it can approach or exceed the complexity of the design task itself. The testbench is full of details and, if constructed using only informal notions of the required behavior and abstraction, then it suffers the same potential problems common to all designs developed to poor specifications.

A fundamental issue here is whether the execution of a second detailed design effort, in parallel with the original design, is a good use of engineering talent. In our view, this is not the preferred approach; instead, comparable effort would be more productively used to instead model the abstract specification level and the abstraction. The benefits of having explicit models for these two entities are unmistakeable. However, another important benefit of these models is the fundamental role they can play in the automatic generation of testbench models, both design and specification.

Figure 2 describes our approach to testbench model construction, and of model construction in general. Not surprisingly, the designer is still responsible for building the design and specification models. However, rather than burdening the designer with testbench construction as well, these are generated automatically using model generation tools. The new block within the figure defines the *abstraction* between the specification and design, and as indicated by the arrows emanating from it, provides the key information used by

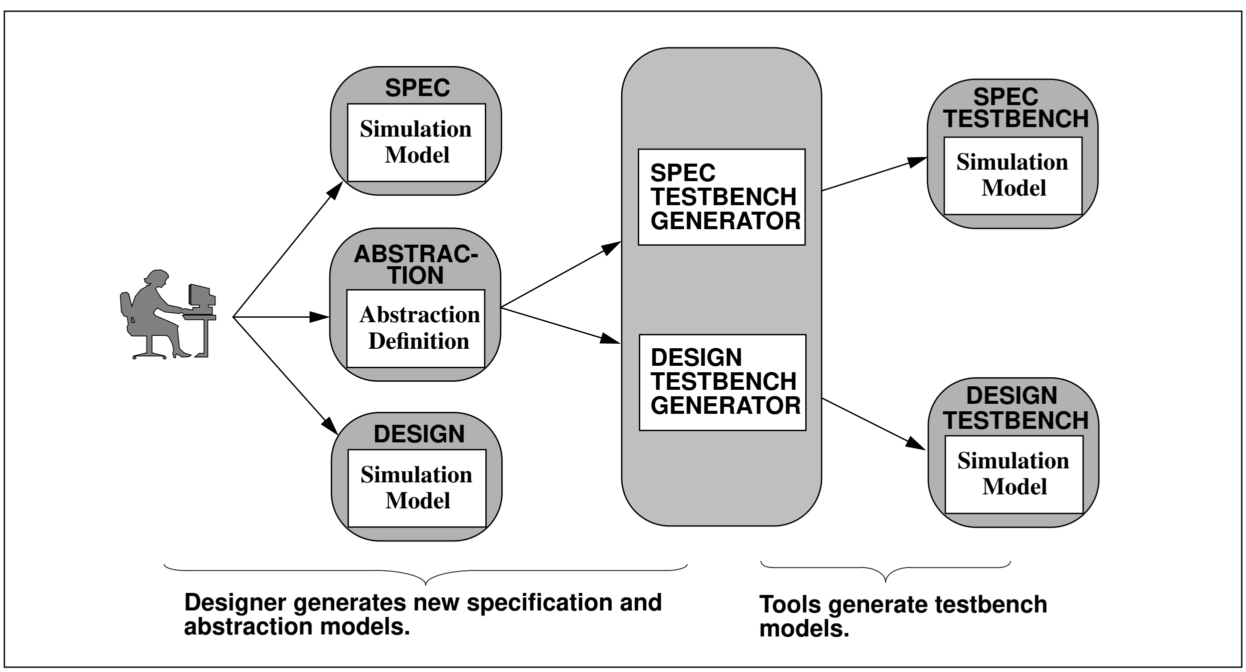

Figure 2: Breakdown of New Model Generation Tasks.

the testbench generators. The abstraction thus constitutes the third, and final, model to be generated by the designer.

We believe that the work described in this section has potential benefits in two areas. First, the improvements in rigor and automation of our approach can have important near-term benefits in the combination of improved test coverage and/or reduced test cost. Greater support for secure abstract-level composition and substantial improvements in documentation are additional benefits.

A second area of potential benefit is in the training of mainstream design and verification engineers in two subjects fundamental to advanced formal verification methods:

(a) abstract-level hardware specification and

(b) interface languages and abstraction.

Our approach presents the design engineer a tradeoff in greatly reduced low-level work in exchange for the explicit definition of a specification model and the abstraction linking it to the design. Having mastered these concepts in a familiar simulation environment, designers would then have a higher skill base from which to pursue other approaches, such as model checking, theorem proving, and so on.

5 Conclusions

Higher-order logic is an extremely versatile platform for constructing interpreter interface languages. Using the standard definitional facilities provided by the HOL theorem proving system, we have developed an interface language that is surprisingly expressive and concise. Because this language is formally embedded within the HOL system, it provides very secure support for modeling interpreter abstraction and conducting interpreter verifications.

In this paper we have introduced new language constructs that provide especially meaningful syntax. For example, the construct **(f For n 'thCountSince t1) At t2** is a 'human-friendly' and more-general form of Melham's **Istimeof f n t2** predicate [Mel90]. In addition, the class of constructs represented by **(f AtLeast n CountsSince t1) At t2** is very useful for modeling hardware (e.g., [Fur94a]), and is, to our knowledge, unique to this work. Many previously-used constructs have been incorporated into our language as well.

Interpreter interface languages support a hardware modeling paradigm that has applications outside of formal theorem proving contexts. Two interpreter models, one an abstract representation of the other, can be implemented using simulation languages, for example. An interface language represents interpreter abstraction just as well in these types of environments. This observation is an important one in that it suggests a promising avenue for research to extend formal methods concepts into mainstream hardware design.

One of the major problems with existing hardware design processes is precisely that they are extremely poor at specifying abstract requirements and then linking these requirements to the detailed design. We believe that research into interface language development for simulation environments has a high potential payoff. For example, rigorous notions of interpreter abstraction and techniques for abstract-level modeling would be most readily accepted if applied to simulation environments directly. Having gotten experience with these concepts in this way, in addition to having learned a temporal-logic interface language, hardware designers would then have a higher skill base from which to pursue other formal approaches.

6 References

[Fur94a] D.A. Fura and A.K. Somani, "Interval-Semantic Component Models and the Efficient Verification of Transaction-Level Circuit Behavior," in T.F. Melham and J.A. Camilleri (eds.), *Higher-Order Logic Theorem Proving and its Applications*, Lecture Notes in Computer Science 859, Springer-Verlag, 1994.

[Fur94b] D.A. Fura, *Abstract Interpreter Modeling and Verification Methods for Embedded Hardware and Fault-Tolerant Systems*, Ph.D. thesis (in preparation), Electrical Engineering Department, University of Washington, 1994.

[Gor93] M.J.C. Gordon and T.F. Melham, *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*, Cambridge University Press, 1993.

[Gra92] "The SECD Microprocessor Specification and Verification," HOL contrib directory, February 1992.

[Han86] F.K. Hanna and N. Daeche, "Specification and Verification using Higher-Order Logic: A Case Study," in G.J. Milne and P.A. Subrahmanyam (eds.), *Formal Aspects of VLSI Design*, Elsevier Science Publishers, 1986, pp. 179–213.

[Her88] J. Herbert, "Formal Verification of Basic Memory Devices," Technical Report No. 124, Computer Laboratory, University of Cambridge, February 1988.

[IEE88] Institute of Electrical and Electronics Engineers, Inc., *IEEE Standard VHDL Language Reference Manual*, March 1988.

[Jas90] A. Jasuja, "temporal.ml," (an HOL formulation of propositional temporal logic), HOL contrib directory, July 1990.

[Joy90] J.J. Joyce, *Multi-Level Verification of Microprocessor-Bases Systems*, Ph.D. thesis and Technical Report No. 195, Computer Laboratory, University of Cambridge, May 1990.

[Joy93] J.J. Joyce and C.H. Seger, "Linking BDD-Based Symbolic Evaluation to Interactive Theorem-Proving," in *Proceedings of the 30th Design Automation Conference*, IEEE Computer Society Press, 1993.

[Man92] Z. Manna and A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems • Specification •*, Springer-Verlag, 1992.

[McM93] K.L. McMillan, *Symbolic Model Checking*, Kluwer Academic Publishers, 1993.

[Mel90] T.F. Melham, *Formalizing Abstraction Mechanisms for Hardware Verification in Higher Order Logic*, Ph.D. thesis and Technical Report No. 201, Computer Laboratory, University of Cambridge, August 1990.

[Mos85] B. Moszkowski, "A Temporal Logic for Multilevel Reasoning about Hardware," *IEEE Computer*, Vol. 18, No. 2, February 1985, pp. 10–19.