A Brief Introduction to Formal Methods *

Paul E. Black Kelly M. Hall Michael D. Jones Trent N. Larson Phillip J. Windley

Laboratory for Applied Logic
Brigham Young University

info-lal@lal.cs.byu.edu

Abstract

As hardware designs grow in size and complexity, current design methods are proving less adequate. Current methods for specification, design, and test are typically empirical or informal, that is, they are based on experience and argument. Formal methods are solidly based on mathematical logic systems and precise rules of inference. Formal methods offer a discipline which complements current methods so designers can successfully meet the demand for high performance systems.

Formal methods covers a broad and diverse set of techniques aimed at improving computer correctness. This paper explains the role of specifications and implementation models in formal methods, and different approaches to proving their correspondence. We refer to excellent overview papers and cite some recent successful examples of using formal methods in hardware design.

Introduction

Current hardware and software designs are orders of magnitude larger and more complex than they have been. It is therefore more difficult to design correct systems using only informal techniques and practices. The term *formal methods* includes a set of techniques based on mathematical foundations and analysis. Formal methods [10] improve computer design by reducing design errors when used as a complement to empirical techniques currently used. This paper provides a brief introduction to formal methods for hardware design.[1] We discuss what they are, describe different methodologies grouped under the heading *formal methods*, and suggest where they can be used successfully. Due to space considerations, the bibliography is not extensive, but it was carefully chosen to provide a good starting place for further exploration.

*This work was sponsored by the National Science Foundation under NSF grant MIP–9412581 and the Department of Defense under contract MDA904-94-C-6115.

[1] This paper focuses exclusively on formal methods in hardware design. Formal methods can also be used in broader system design, including software, but such discussion is beyond the scope of this paper.

What are Formal Methods?

Formal methods are an analytical approach relying on mathematical models for excluding design errors in hardware. Other approaches to design fault exclusion, such as simulation, are empirical in nature. The chief benefit of analytical techniques is that they offer 100% coverage of the design space. That is, with a precise mathematical model, one can reason about all possible cases. The chief drawback is the difficulty of building models and conducting analysis. The precise nature of formal methods precludes informal hand waving to dismiss difficult, extreme cases.

All formal methods involve one or more of the following three parts:

1. a mathematical model of the design's intended behavior or properties, called the *specification,*

2. a mathematical model of the design's structure, called the *implementation model,* or more briefly, the implementation, and

3. mathematical expressions stating relationships between the models established using analysis (proof) to demonstrate that the relations hold.

Formal methods begin with a specification, an implementation model, and a mathematical expression stating the relationship between them. They finish by demonstrating the relationship via precisely defined rules. However, formal methods need not include all three aspects. Benefits accrue from simply writing a formal specification which then serves as an unambiguous reference for implementation, simulation, and testing.

A. Simple Example of Formal Methods

As an example of the activities and models discussed above, we might specify the behavior of an exclusive–or gate with the following mathematical formula:

$$\vdash_{def} \texttt{xor2\_spec a b out} = (\texttt{out} = \neg(\texttt{a} = \texttt{b}))$$

That is, the behavior of an exclusive–or relates inputs, a and b, and the output, out. Note that the above formula can easily be assigned a rigorously defined meaning. The implementation model could be described using a netlist:

```
MODULE out .xor2_imp a b;
       BEGIN
       p    .nand2  a b ;
       q    .nand2  a p ;
       r    .nand2  p b ;
       out  .nand2  q r ;
       END;
```

This implementation is four interconnected `NAND` gates. In addition, we must have rigorous definitions of the meaning of `MODULE`, `.nand2`, `BEGIN`, etc., so that the above implementation model also has an unambiguous meaning.

We wish to show that the implementation satisfies the specification. We can express this with the mathematical relation *implies* and express rigorous definitions of the netlist (not given here for brevity) as a function `INTERP` in the following manner :

```
⊢ ∀ a b out. INTERP(out .xor2_imp a b) ⇒
             xor2_spec a b out
```

One can also read the formula as, for all `a`, `b` and `out`, the interpretation of an `XOR2` implementation (as defined above) on `a, b` and `out` implies the `XOR2` specification (also defined above) on `a, b` and `out`. Using mathematical analysis and the definitions of `.xor2_imp`, `xor2_spec`, and `INTERP`, we can prove that the implementation satisfies the specification.

Notice that the relationship covers *all* values of the inputs and output (`a`, `b`, and `out`), not just some test values. Of course, in this simple example an exhaustive simulation is trivial, but many formal methods can be applied to circuits with 10^{100} states or more and still show that the relationship holds for all possibilities.

How Do I Put Formal Methods to Work?

Various formalisms and techniques are applicable to each part of the process described in the previous section. To write a formal specification, one must make choices about which formalism to use (first order logic, higher order logic, temporal logic, state machines, automata, trace specifications, etc.) and the kinds of criteria to specify (functional correctness, liveness, safety, timing, and so on). To model a circuit, one must decide which level of abstraction (gate level, switch level, circuit level, register-transfer level, or higher) is appropriate as well as which formalism to use (first order logic, higher order logic, automata, etc.). The relationship of implementation and specification may be equivalence, implication, etc. How one handles each of the three parts forms a taxonomy of formal methods tools and techniques [3].

A. The Specification

Writing a specification for a design is perhaps the most difficult aspect of the formal methods process. Formal specifications require the designer to clearly, concisely, and unambiguously state what a circuit must do. To be of any benefit, the specification must be an abstract representation of the implementation. That is, it should state *what* a circuit must do, not *how*. The abstractions may be any combination of structural (an ALU instead of gates), data (numbers instead of bit vectors), temporal (instruction cycles instead of clock cycles), or behavioral (a page from memory is saved to disk instead of which page is saved to which cylinder). Specifications may be quite comprehensive, or they may include relatively few fundamental requirements such as a request is eventually granted or two communicating devices never deadlock. Specifications can also indicate timing properties, load characteristics, and other properties of the device.

The idea of formal methods is to show that the implementation meets the specification; but how does one ensure that the *specification* is correct? Ultimately it must be *validated* by the designers: they must examine the specification and decide that it expresses what they want. Higher level abstractions help by making it easier to state desired properties and behaviors. More powerful representations can more easily and concisely express the designer's desires. A specification of a few fundamental properties may be easy to judge correct, but leaves other important properties only informally specified. Some representations are executable, allowing designers to validate the specification by simulation in addition to review.

One of the most important choices to make is the level of abstraction in the specification. Higher level abstractions tend to allow more concise specifications, since less detail is included. Abstraction causes the specification to be more easily modified and validated. On the other hand, an abstract specification is more difficult to relate to the implementation. Multi-level verification treats the one level's specification as the next higher level's implementation. Thus several simple abstractions can be independently verified to yield the overall proof.

Related to the level of abstraction is the expressiveness of the language. A simple language, such as state machines or first order logic, is easy to reason about — in fact, many simple languages are decidable: they have completely automatic algorithms for calculating the correctness of statements. More expressive languages, such as higher-order logic, can more concisely express a wide range of specifications, but they are more difficult to reason about.

More abstract and expressive languages are more powerful in the long run, but tend to require more initial investment since they are more mathematical and less like representations with which designers are familiar.

B. The Implementation Model

Creating a model of the implementation is a standard task for hardware designers. Implementation models are similar to simulation models in use by designers now. An implementation model may be extracted from the simulation model, or, potentially, the same model may be used for both verification and simulation. The implementation model *must* have a well defined interpretation or meaning. A model with simple primitives is easier to reason about, but is a poor representation of the circuit. A more detailed model is a better representation of the circuit, but it is more difficult to use in a verification.

How does one ensure that the implementation model actually represents the physical device? As with the specification, validating the implementation can not be done by machine. Since the model only represents certain characteristics of the device, the final design must be checked to ensure that it has those characteristics.

Few formal methods tools accept models written for standard simulation tools without significant syntactic changes to the model. Most tools require a completely new model expressed in a different modeling framework. Thus, a designer often must construct multiple models of their circuit, one for each design tool (simulator, formal methods tool, etc.). Multiple versions raise the cost of design maintenance and can lead to version skew problems. Current research in formal methods is aimed at using standard HDLs for implementation modeling and providing increased simulation capability.

C. Relating the Implementation and the Specification

There are several methods currently being used for relating implementations to specifications. These include theorem proving, model checking, equivalence checking, and language containment. Among these, the most commonly used are model checking and theorem proving.

In the model-checking approach, the specification is expressed as a formula in temporal logic. Such logics make statements about a world that changes through time, and they allow reasoning about dynamically changing situations. Implementation models are usually in the form of state transition graphs. They can be compared with the specification automatically. The system may verify that the models are valid, or it may provide counter-examples for any specifications falsified by the implementation.

Since model checking typically uses state-based hardware descriptions, it is better suited for checking control structures, as it expresses things such as concurrency and synchronization. Also, by modeling a particular kind of logic and world, model checkers aren't excessively complex. Of course, this means that they are not readily used on other types of problems.

As an example, the well known model checker SMV [6] uses specifications written in the temporal logic CTL and implementation models written in a VHDL-like language for creating state machines.

Theorem proving, in contrast, is a more interactive technique. When one uses a theorem prover to verify hardware the usual process is to design a specification and implementation as logic descriptions first-order predicate logic, higher-order logic, etc. The designer then guides the proof assistant tool through rigorous proof steps showing that the implementation model satisfies the logical specification. The level of interaction required of the user varies widely between theorem prover tools: some tools demand much detail but offer great flexibility (e.g., HOL); other tools are more automatic at the expense of flexibility (e.g., PVS, NQTHM).

Theorem provers ultimately rely on the designer to create an appropriate model of the hardware, and even to guide the system (sometimes explicitly) along the path to a proof. This can be a complex process, but theorem provers are very general and can be employed in a wide range of applications. Theorem provers are not as useful for reasoning about temporal aspects of hardware. But theorem provers are well suited for hierarchical methods of development (due to their abstraction mechanisms) as well as reasoning about functional specifications and parameterized descriptions.

The NQTHM theorem prover uses Boyer-Moore quantifier-free first order logic (with equality) to represent both specifications and implementation models [4]. The HOL theorem prover [9] uses higher-order logic to produce a flexible, but demanding environment for creating specification and implementation models.

What Will Formal Methods Do?

Formal methods will ensure that an implementation meets a specification, but they will not guarantee that the final product will always operate perfectly. What formal methods can do is limited by the philosophical limits on what can be proven, informally defined description languages and extra-logical factors. These limitations ought to be kept in mind when describing and discussing the results of a formal verification project. A more complete discussion of what formal methods will and will not do can be found in [2].

There are aspects of the design process to which the notions of specification and implementation simply do not apply. Specifications can not convey design intentions and implementation models can not describe physical properties. Consequently, the verifier is forced to choose a level of abstraction for the specification and a sufficiently concrete level for the implementation model. These decisions are part of the process, but should be explicitly stated when making a claim about verification. For example, claiming that a microprocessor has been verified at the gate-level implementation to meet a functional block specification more completely conveys the scope of the verification than simply stating that the processor has been verified.

In practice, the designer, verifier and each manufacturer use different, informally defined languages to describe the design. Often, the translations between these languages requires a combination of experience, intuition and luck. This is especially true of low-level design descriptions which may consist of nothing more than annotated diagrams and a few paragraphs of text. Without mathematically precise definitions of the design-description language, it is impossible to know if the verifier's interpretation of the design is the same as the designer and if the design manufactured is the same as the one described by the verifier. These gaps can be bridged by using a common, formally defined language at all three levels. Work on these types of languages is underway at Computational Logic, Inc. and Brigham Young University [11].

Claims about verified devices, especially in safety or security critical applications, should be strictly limited to factors covered by the logic. Faulty communication, social hierarchies, political climates, and so forth are usually not covered in the verification process. For example, verification demonstrating that the low-level model of a chip prevents unauthorized users from accessing data does not guarantee that a passer-by could not read sensitive data on a monitor. Consideration of these factors ought to temper broad guarantees about verified devices.

Formal methods have been successfully used in commercial and academic designs. We mention a few to suggest the wide applicability and utility of formal methods.

Johnson, Miner, and Camilleri[5] compare several formal tools by implementing a simple circuit in each of them. They "contrast how the underlying formalisms influence one's perspective on design and verification."

Windley and Coe[8] verified the correctness of a simple pipelined microprocessor using HOL, and Srivas and Miller[7] report the formal verification of a commercial microprocessor. Bainbridge, Camilleri, and Fleming[1] relate verifying memory protocols in an industrial setting. They point out formal methods can be used even with a short *time to market.*

Conclusion

Formal methods are a useful addition to the hardware design process. To make use of formal methods, first a specification must be written which expresses the design criteria, then an implementation model must be written or captured which represents the design. Formal verification demonstrates that an implementation model meets a specification for *all* cases. However formal methods are not a *silver bullet* which prevents all errors. Rather formal methods are a complement to good design methodology and testing. The chief benefit of formal methods is not the final *true* from the tool, but rather the process that is required to get the final result.

While the wide variety of specification and implementation modeling techniques and the many ways of relating them can make the choice of how and when to apply formal methods sound daunting, in fact there are relatively few tools to choose from. Of these tools, the best criteria for choosing is how they handle the task of relating the implementation model and the specification. Once a particular tool is chosen, many of the options for creating the implementation model and specification are eliminated.

References

[1] Simon Bainbridge, Albert Camilleri, and Roger Fleming, *Theorem Proving as an Industrial Tool for System Level Design*, in V. Stavridou, T. F. Melham, and R. T. Boute (eds.) *Theorem Provers in Circuit Design*, North-Holland, 1992.

[2] Avra Cohn, *The Notion of Proof in Hardware Verification*, Journal of Automated Reasoning, Vol 5, No 4, pp. 127–139, 1989.

[3] Aarti Gupta, *Formal Hardware Verification Methods: A Survey*, Formal Methods in System Design, Vol. 1, 1992, pp. 151–238, Kluwer Academic Publishers.

[4] William A. Hunt, Jr., *Microprocessor Design Verification*, Journal of Automated Reasoning, Vol 5, No 4, pp. 429–460, 1989.

[5] Steven D. Johnson, Paul S. Miner, and Albert Camilleri *Studies of the Single Pulser in Various Reasoning Systems*, in R. Kumar and T. Kropf (eds.), *Theorem Provers in Circuit Design*, Springer, 1994.

[6] Kenneth L. McMillan, *Symbolic Model Checking, An Approach to the State Explosion Problem*, Ph.D. thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, 1992.

[7] Mandayam Srivas and Steven P. Miller, *Applying Formal Verification to a Commercial Microprocessor, Proceedings of the 1995 IFIP International Conference on Computer Hardware Description Languages.*

[8] Phillip J. Windley and Michael Coe, *A Correctness Model for Pipelined Microprocessors*, in T. Kropf and R. Kumar (eds.), *Proceedings of the 1994 Conference on Theorem Provers in Circuit Design.*

[9] *Description of the HOL Theorem Proving System*, (URL: http://lal.cs.byu.edu/lal/hol-desc.html)

[10] *Other Formal Methods Sites page*, (URL: http://lal.cs.byu.edu/otherFMSites.html)

[11] *The Laboratory for Applied Logic homepage*, (URL: http://lal.cs.byu.edu/)