Using Formal Methods to Verify Complex Designs

IBM Haifa Research Lab
The IBM center of competence for formal verification
Table of Contents

How Can You Be Sure Your Design Works as Intended? .........................3
Why Formal Verification? .................................................................4
   Current Status of Formal Verification .............................................4
   PSL - A Standard Property Specification Language .............................5
What to Look for in an FV Solution ....................................................6
RuleBase PE from IBM ......................................................................7
   Contact Information .......................................................................9
References ......................................................................................10
How Can You Be Sure Your Design Works as Intended?

Designing a robust hardware system is a challenging task. As market demands grow and technology breakthroughs are introduced, systems are becoming more and more complex. Checking such large systems is likewise becoming more difficult. The most common way to check the correctness of a design is by simulation. Simulating a design involves compiling it into a general binary form familiar to the simulator, providing the input vectors for each test to be performed, and starting the simulation. The output is then investigated to find any flaws in the design.

For many years, simulation-based verification has been used to verify hardware designs before they are sent to be manufactured. Simulation-based verification is easy to do and easy to comprehend. The engineer (or a test case generator) simply creates test cases as inputs for the design and runs the simulator. But is this enough?

Simulation based verification has many disadvantages. The most prominent drawback is that it does not provide complete verification. A hardware design can be thought of as one big state machine. While a state machine has a finite number of states, the length of paths through the state machine is unbounded. Thus, there are an infinite number of such paths, and so checking all possible paths by simulation would require an infinite amount of time. Only a small fraction of the state space is explored using simulation; consequently, hidden bugs often make their way into the final hardware being manufactured. Another disadvantage of simulation-based verification is the need to write the inputs for the simulation runs. Random simulations do not
necessarily cover a meaningful portion of the state space, thus carefully crafted manual tests and/or intelligent test generation is needed.

**Why Formal Verification?**

Formal verification uses mathematical formal methods to prove or disprove the correctness of a system’s design with respect to formal specifications expressed as properties in a Property Specification Language such as PSL. The systems are verified by providing a formal proof on an abstract mathematical model of the system. Over the past 15 years, a new technology called ‘model checking’ has emerged in the design verification area. Model checking is a way to systematically perform exhaustive exploration of the mathematical model, thus achieving a higher level of completeness.

Basically, formal verification takes the verification process and turns it around. Instead of supplying inputs and expecting to see the desired behavior—as is done with simulation—users specify the desired behavior and the formal verification tool makes sure they have covered all possible inputs. Of course, not all possible inputs are useful or desired. Some of the inputs may be illegal for the system. To overcome this problem, users also specify the environment that drives the system under test. The environment is specified in a non-deterministic manner—thereby covering all possible combinations—which is very different from describing each possible input sequence.

The completeness of formal verification comes from its exhaustive coverage of all possible states of the system.

By enabling users to provide those properties that must hold true, formal verification allows them to concentrate on specifying what they expect from their design, and then check it for correctness.

**Current Status of Formal Verification**

Many vendors now offer formal verification tools. With some vendors the tool is part of a complete design solution, whereas others offer a formal verification tool as their main application. Among the high-end tools from large vendors, you can also find
some free tools that are useful for beginners who want to become more familiar with the field.

**PSL - A Standard Property Specification Language**

Users need some sort of formal language in order to specify the behavior of a system over time. Take, for example, the following behavior:

The ack signal will be sent within three to five cycles after the req signal is asserted, unless the request is cancelled.

This specification fragment contains a notion of time, since some event must occur after the first event took place. Users need a way to express this property for the formal verification system.

Decades ago, philosophers and mathematicians began research into the field of temporal logic. Special languages to describe temporal logic have been evolving over the past 50 years. More recently, the Property Specification Language (PSL) became accepted as a worldwide standard.

PSL is a temporal language that attained IEEE standardization in 2005 ([IEEE 1850™-2005](https://standards.ieee.org/)). The PSL language has operators that describe the ordering of events in time. Take, for example, the following PSL code:

`assert always (req -> next ack)`

This simple code states that whenever the signal `req` is asserted, the `ack` signal should be asserted on the next cycle. Tools that can handle PSL should be able to process this code and have the proper algorithms to check if it holds for the design under test. The term “holds” is used to indicate that a certain assertion is true for the design under test.

The PSL language has many different ways to express temporal specifications. In addition to the temporal operators such as `always` and `next`, PSL offers a construct called Sequential Extended Regular Expressions, or simply SERE. A SERE may remind you of regular expressions:

`assert never {write ; write}!;`
The PSL statement above states that the system should never process two consecutive writes.

```psl
assert never { req; (!ack)[^]; req }!
```

This PSL statement states that we should never see a sequence of cycles in which there are two assertions of `req` without an intervening assertion of `ack`.

### What to Look for in an FV Solution

Today you can find many different vendors selling formal verification tools and solutions. There are some key points to consider when evaluating what tool or solution is best for your needs.

- **Capacity** – What is the capacity that can be handled by the formal verification tools you are considering and what is the size of the designs you are about to explore? Formal verification tools have a limited capacity and can work on state spaces in the order of no more than 10K state variables (after reductions). This issue may influence the way you code your design, so that you choose to write smaller basic units (modules or entities) that can later be used to facilitate the verification process.

- **Usability** – A formal verification tool should aid in all stages of the verification process. Starting from writing the specification code, compiling the design, parsing and debugging the code, and finally verifying the assertions. An effective tool will support all these stages and allow the verification engineer to concentrate on the essence of the design rather than spend time searching for the correct flag.

- **Well-known specification language** – Standard specification languages are very important for code reuse. Once specification code is written, it can later be used in other projects and other tools, provided it is written in a well known language. A good tool should support a standard specification language.

- **Variety of algorithms for proof** – Model checking has many different approaches and algorithms. This field is
constantly evolving with new breakthroughs and proof engines. A good formal verification tool should have a variety of algorithms the user can choose from in order to tackle various problems.

- **Environment modeling languages** – HDL code is used to specify the driving model (also called the environment) for the verification process. A good tool will support popular HDL languages for modeling, and especially the team’s HDL design language. This way, the verification engineer or the designer can use a language they are familiar with when using the verification tool.

- **Diversity of design languages** – Several popular hardware design languages dominate the market. The formal verification tools should be able to deal with most of them, especially as design input. A good tool should support the language you use for your designs.

- **Parallelism** – As computer arrays become more common, using parallel computing abilities offers a significant advantage. Formal verification consumes a considerable amount of time and computing resources. A good tool will incorporate parallelism, thus making more efficient use of the system’s computing power.

- **Reliable and field proven tools** – Verifying hardware is a crucial step towards a finalized hardware project. Companies can’t afford to use a verification tool that will fall short and impair the completeness of this process. A good verification tool is one that has proven itself in the market and has participated in successful projects.

- **Technical support** – Occasionally some help is required to accomplish a good Formal Verification project. A good verification tool will be backed by a strong technical team that can address problems and questions.

**RuleBase PE from IBM**

RuleBase PE is a verification platform that allows engineers to harness the power of model checking and parallel computing,
thereby enabling them to rapidly verify complex chip designs and ensure their new devices work the way they should. Using RuleBase PE, a design is validated against its functional specification, as captured by user-specified assertions. The platform is based on parallel formal and semi-formal verification algorithms that were developed at the IBM Research Lab in Haifa, Israel.

RuleBase PE is an industrial-strength tool that gives designers and verification engineers an upper hand when it comes to finding evasive design bugs throughout the design cycle. It can be used to verify large-scale designs in a fraction of the time required by other formal verification techniques.

IBM first developed RuleBase in 1993 and successfully used it to verify internal hardware developments. RuleBase originally used the temporal logic Sugar as its specification language. Over the years, this language evolved and eventually served as the basis for the PSL standard.

RuleBase PE offers these vital features:

1. RuleBase PE can handle large designs due to its very efficient reduction algorithms. The reduction algorithms in RuleBase PE can shrink the size of the verified model by an order of magnitude.

2. RuleBase PE has a simple and easy-to-use GUI. Even the most inexperienced user can become productive in no time, focusing on what matters most – formulating the assertions in PSL.

3. RuleBase PE offers excellent support for the PSL language, since PSL is the native temporal language developed for use in RuleBase PE.

4. RuleBase PE has six different formal verification engines. Each engine is based upon a different algorithmic approach: BDD based engines together with SAT based engines and semi-formal engines contribute proving and bug hunting capabilities for large designs.

5. RuleBase PE supports two flavors of PSL to describe the environment: Verilog, the standard language IEEE 1364.2001; GDL (General Description Language) is a simple hardware
design language invented especially for RuleBase PE. It is strong and expressive, yet extremely easy to learn.

6. RuleBase PE supports Verilog, VHDL and a mix of these languages as design inputs languages.

7. RuleBase PE supports both fine grain and coarse grain parallelism; both work with most common load balancing systems. With coarse grain, different verification engines can be sent to different machines; when the best engine for the task finds the solution, it cancels all other engines. Fine grain parallelism is used within certain engines that can split their task among several processors.

8. RuleBase PE has more than 10 years of field-proven verification experience, both inside and outside IBM. In IBM, RuleBase was used to verify:
   - pSeries (SP, ASCI, Power 4,5,6)
   - iSeries (AS/400)
   - zSeries (Freeway, S/390)
   - xSeries (Netfinity)
   - Game processors (Cell, Xbox)
   - MD cores (PCI, USB, Ethernet, ...)

9. IBM provides RuleBase PE users with access to the top verification experts in IBM—the world’s largest IT company. RuleBase PE is backed by a strong team of experts who have a deep understanding of formal verification, a history as pioneers in the forefront of verification technology, and a worldwide reputation for excellence.

### Contact Information

For more information on the IBM RuleBase PE offering, please contact:

- Sivan Rabinovich  
  IBM Haifa Research Lab  
  Tel: +972-4-8281108  
  Email: sivanr@il.ibm.com
• Sharon Keidar-Barner  
  IBM Haifa Research Lab  
  Tel: +972-4-8296157  
  Email: sharon@il.ibm.com

Visit our website:  
http://www.haifa.il.ibm.com/projects/verification/RB_Homepage

References

• C. Eisner, D. Fisman "A practical introduction to PSL", Springer 2006,  

• Beer, S. Ben-David, C. Eisner, A. Landver, "RuleBase: an Industry-  
  Oriented Formal Verification Tool", DAC’96.

• S. Ben-David, C. Eisner, D. Geist, Y Wolfsthal, "Model Checking at IBM",  
  101-108.
Notices

International Business Machines Corporation provides this publication “as is” without warranty of any kind, either express or implied. Some states do not allow disclaimer of express or implied warranties in certain transactions; therefore this statement may not apply to you.

This publication may contain technical inaccuracies or typographical errors. While every precaution has been taken in the preparation of this document, the publisher and author assume no responsibility for errors or omissions. Nor is any liability assumed for damages resulting from the use of the information contained herein. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/ or changes in the product(s) and/or the program(s) described in this publication at any time.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Director of Commercial Relations, IBM Corporation.

All trademarks or service marks are the property of their respective owners.

© Copyright IBM Corporation 2007. All rights reserved.