CREATIVE ASSERTION AND CONSTRAINT METHODS FOR
FORMAL DESIGN VERIFICATION

Joseph Richards
SGI, High Performance Systems Development
Mountain View, CA
richards@sgi.com

Abstract

The challenges involved in verifying a complex ASIC chip become daunting for a limited team of design and verification engineers. The recent use of standardized assertion languages and static property checking tools has allowed us to break the normal bounds of functional verification. Assertions provide much-needed visibility into the inner workings of a design, and formal verification is a means of verifying key micro architectural functionality. A hierarchical approach to assertion writing aids verification when using a formal tool. Shared RTL can be verified as a single entity when its desired capabilities are defined and proven with a formal tool. Using these formal verification methods along with standard simulation based methods provides greater confidence as a design progresses toward the tapeout deadline.

Introduction

Design verification is becoming an increasingly important phase of the design cycle as product complexity increases. The resources and time needed to completely verify a complex design are now overcoming that which is needed for the actual logic and physical design, therefore new methods must be explored to ease this burden. Two newer methods of functional verification, assertion-based verification and formal verification, are becoming possible candidates as solutions to the growing verification problem. Recent efforts at SGI have explored these methods and their integration into the standard verification flow.

Assertion-based verification has been used in many previous design projects at SGI as a helpful means of verification. An in-house assertion library was developed for use in the simulation environment. Assertions are design modules that can be used to verify specific design properties[Bening and Foster]. Assertions are especially helpful in verifying key micro-architectural concepts such as state machine exploration, counter movement, and fifo overflow and underflow conditions. They can easily be placed within the RTL code by designers and provide extremely meaningful diagnostic information to verification engineers. This level of visibility is becoming ever more important as the size and complexity of a design increases.

Formal verification has become a widely accepted method for equivalency checking between gate and RTL designs, but its more interesting application of functional verification has not yet become a fully mainstream method. Its capabilities have been proven in the verification of complex architectural algorithms such as cache coherency and bus protocols. SGI has long been a user of the SMV model checker [McMillan] in verifying its complex cache coherency protocols, and this has given us great confidence in the capabilities of formal methods in the realm of functional
verification. Formal tools that check the equivalency between RTL and some other design implementation can be termed formal equivalency checking tools, while formal tools used in the realm of functional verification we term formal static property checking tools. Throughout the rest of the text the term formal/formal verification tool implies a formal static property checking tool.

With our previous experience with assertion-based and formal verification methods, it seemed appropriate to push these two techniques further into our normal realm of design verification. The recent advent of more user-friendly formal static property checking tools has allowed us to integrate both assertions and formal verification into our standard flow of RTL design verification.

The Open Verification Library [OVL] was chosen as the assertion library of choice for many future projects because of its standardization and cross platform usability. The library gives us a base set of assertions that can be used on their own, or combined with temporary logic to implement more complicated monitors. Verplex’s BlackTie Functional Checker [Verplex] was selected as the tool to formally prove our assertion properties. The BlackTie tool has the ability to automatically extract static properties from the design and prove them formally, but the tools true verification capability is best shown through the proving of user-defined assertions.

**Formal Assertion-Based Verification**

With the assertion library and formal tool chosen, we started to explore the capabilities of these new verification techniques and define methodologies to aid our ASIC development. The first step was to understand the types of assertions available and becoming knowledgeable about the formal tool. In our initial uses of formal assertion-based verification, we found that this method was much different than the conventional simulation verification that we had primarily been using.

The key difference between the use of assertion-based verification and conventional simulation-driven techniques is the fact that assertions are very closely related to the code itself. Most verification efforts take a black box approach, verifying key capabilities and functionality of the design without knowing the actual implementation details[Foster and Coelho]. For assertion-based verification, a white box approach must be taken to verify the implementation of the design.

Assertion-based verification using a formal checking tool requires a very different approach than standard verification methods. In standard transaction-based test bench environments, real and valid functional tests can be created easily and results seen visually. In assertion-based verification, the design must be broken down into small, definitive properties. Transaction-based verification is intuitive in nature, but limited in scope of coverage, while assertion verification using a formal tool takes more forethought, but provides possibly exhaustive coverage.

In writing assertions it’s important to understand the two classifications of assertion, a property and a constraint [Foster and Coelho]. A constraint is an assertion that bounds the input stimulus to permissible operating values. Constraints become extremely important in formal verification as the initial assumption by the tool is that EVERY input stimulus is permissible. A property is an assertion that defines an expected behavior in response to all permissible inputs. Property assertions are defined to insure that a design is functionally correct under all permissible input conditions.
The true value of formal verification tool lies in its ability to exhaustively search the state space of a design, revealing bugs in corner cases that would have been very difficult to find using traditional simulation-based simulation. This exhaustive formal verification of a design may be the greatest benefit of the tool, but it can also be one of the first hindrances.

Giving the tool complete freedom to explore every possible condition is truly valuable for a large number of verification problems. There are many design properties that must remain valid for every possible situation. But there are also a great number of design properties that can only be verified using assumptions about the behavior of the surrounding logic. For these instances, constraints are the key to getting the formal tool to verify the design properly.

Assertion writing efforts were initiated immediately upon the start of RTL coding for SGI’s latest system design project. As RTL was developed according to the specification of the design, assertions were also written according to the spec and the developing RTL. An intimate knowledge of the RTL was needed to effectively craft assertion that verified key functionality.

The Blacktie Functional Verification tool was used immediately with the developing code and newly written assertions. One of the greatest advantages of using a formal property-checking tool in such a manner is that there is no need to develop a testbench environment to drive the verification. The tool can instantly read in Verilog code and the OVL library and produce meaningful results. These results can be fed immediately back into the development of the RTL. This allows for the creation of far more robust code earlier than in conventional development cycles.

**Hierarchical Constraint Approach**

As was mentioned before, the use of constraints is key to effectively using a formal tool to verify design properties. Without constraints the formal tool will exhaustively explore every possible input condition, even those that in normal operation are completely illegal. These illegal conditions must be alleviated with constraints for the property assertions to actually provide valuable verification information.

Separately verifying smaller blocks of a larger design on their own is vital when using the formal tool, as it is difficult for formal tools to handle large and complex designs. When verifying these smaller blocks of logic, it is important to understand when and how to constrain the formal tool. Any logic that relies on having timely, legal input sequences to function correctly needs to be constrained properly when being verified in isolation.

There are two areas of concern when developing constraints for a block of logic. The first and most important concern is that one may over-constrain the design, thus not giving the formal tool the ability to verify every possible corner case. When one writes a constraint you want to be sure that you eliminate only the cases that are impossible, and not also eliminate those that are actually possible. The second concern is how these constraints will be applied throughout the design hierarchy. We developed coding methods to try and alleviate these concerns.

We used three methods to help in alleviating over-constraint. The first method was writing only boundary constraints. All signals used in a constraint could only be inputs to the design under test. No constraint could use or constrain internal signals. This insured that we only constrained the tool’s ability to apply invalid stimulus and not
constrain the internal logic. Second, no constraint was written until it was absolutely needed. One could not write a constraint until it is seen that a certain property is failing because of invalid inputs. At that point a constraint could be written only to fix that failing property. Lastly, boundary constraints were written so that they were also applicable as regular property assertions at a higher level. This insured that one is not writing invalid constraints that are illegal when all blocks of logic are assembled into the total design.

The second concern we had with constraints is how they would be applied at a higher level. Because assertions were written and embedded close to the RTL itself, all constraints that were written at a lower level would also appear when verifying the design at a higher level. If we keep the rule that all constraints can be viewed as property assertions at a higher level, there should be no problem when trying to verify lower level constraints as design properties. As discussed before, this helped in alleviating over-constraint, but it also helped in keeping a consistent flow of assertions throughout the design hierarchy.

To effectively use constraints throughout our design hierarchy, we needed to implement a consistent method to create and name constraints. Using a consistent method throughout the design helped in the integration of smaller modules of logic into blocks, super-blocks and finally the full chip. Using a specific naming convention allowed us to flip switches within the BlackTie tool to add, remove or modify constraints and assertions for different runs of the tool. This allowed us to have an OVL assertion act as a constraint at one level of the design, and then use the same instance as a true design property at a higher level.

![Hierarchy method for assertion and constraint placement](image)

**Figure 1:** Hierarchy method for assertion and constraint placement
The naming convention we chose included the level of the design hierarchy in which an assertion/constraint was sitting. This naming convention was used for both assertions and constraints in a design. Lower level constraints that were used to constrain the design when verifying at that low level, could then be verified as properties when running at a higher level. They could even be turned off completely when verifying at a much higher level, since they were proven conclusively at a lower level.

Example 1: Property and constraint written for lowest level of hierarchy

```
assert_proposition example_a1(...);  // Property
assert_never example_c1(...);       // Constraint
```

Example 2: Verifying lowest level in formal tool

```
> add assertion constraint *_c1    // constrain tool
> add static property *_a1         // add properties
> prove                           // prove formally
```

Example 3: Verifying higher level in formal tool

```
> add assertion constraint *_c2    // constrain tool
> add static_property *_a2         // add properties
> add static_property *_c1         // add lower level constraints
> prove                           // prove formally
```

As you can see, when running the formal tool at a higher level, we are not only verifying the assertion properties that were defined for that level, but also verifying the lower level constraints that had now become design properties.

The same method can be used to turn assertions on and off at different levels. This became important because of the formal tool’s difficulty in verifying lower level assertions in a much larger design. The functionality found within a module should be verified as a single entity in the formal tool. Proper interaction between modules can then be verified at a higher level of hierarchy.

There is also the issue of how all these properties and constraints will be included when running the design through standard simulations. There should be no problem in running any of the written assertions through simulations, as they should be applicable for any type of verification method. If constraints are written in the correct manner, in that they become property assertions at some higher level, then there should also be no problem in running them in standard simulation environments.

With the use of this hierarchical property and constraint approach, we were successful in effectively verifying properties throughout our design using the formal tool. Constraints written for smaller modules of the design could be instantly proven as properties when pulling logic pieces together into a larger design. The majority of the new RTL design was verified using this bottom up approach to hierarchical assertion writing.
Shared RTL Approach

Shared RTL is a prime target for assertion-based verification because of its clearly defined properties and its re-use throughout a design. A shared piece of code can be verified once and can then be placed confidently in multiple parts of the design.

One particular example of shared RTL was our implementation of a 144-bit ECC algorithm. This ECC generating and checking code was to be used in various parts of the design. The actual algorithm used in the ECC code had been verified exhaustively through software methods, but the comprehensive verification of the RTL implementation was still needed.

To verify the implementation of the ECC algorithm, a stand-alone testbench was developed. Because this piece of code was being instantiated multiple places throughout the chip, it was ideal to verify the code once as a single entity to reduce redundant verification efforts. The ECC code was implemented in the form of two blocks, an ECC generation block and an ECC checking block. The most efficient way to verify the two blocks was by combining them into a single testbench. The goal was to verify the complete implementation of the algorithm, and since both blocks together comprised the algorithm, it made sense to verify them together.

To exhaustively verify the ECC implementation, all possible combinations of data inputs and error injection had to be applied. The design needed to be tested to see if the data was corrected or the error was detected depending on the error injection. These criteria were expressed as a series of assertion properties. The formal tool was then given the ability to manipulate all data and error injection bits to exhaustively prove these properties.

The testbench was a verilog wrapper that instantiated the two ECC blocks. The inputs of the generate block were pulled up as the inputs to the testbench. The outputs of the checking block were pulled out as the outputs of the testbench. The outputs of the generate block and the inputs of the checking block were tied together with a single level of XOR logic between them. This XOR logic enabled injection of error on any given bit between the two blocks. The inputs to the error injection XOR logic were then pulled out as inputs to the testbench. With the available inputs and outputs of the testbench, the formal tool was able to exhaustively prove the algorithm’s desired capabilities.
Assertions were written to verify correction of single bit error correction and all other correction and detection capabilities of this algorithm. A simple assertion to check single bit correction could be written as follows.

Example 4:

```
assert_proposition (~reset, <single bit error condition>,
data_in == data_out);
```

Multiple iterations of the same assertion were written and proved. This allowed for the complete verification of the correction of every single bit error. Other assertions were written to verify the remaining capabilities of the ECC algorithm. After running of the formal tool for approximately 24 hours, all capabilities of the 144-bit ECC implementation were completely and exhaustively verified.

**Future Challenges**

Through our experience with formal assertion-based verification we found many benefits of this verification method and solved many problems with its day-to-day usage. But there remain many challenges in making formal assertion-based verification a mainstream technique.

An important challenge is to increase the involvement of designers in the writing of property and constraint assertions. Designers must be convinced of the effectiveness of this method. It must be emphasized that the small effort taken in the creation of assertions during the development of code will greatly shorten the development, verification and debug cycles of the design. Increasing the ease of use of formal tools will also aid in the acceptance of assertion writing by designers.

The current capabilities of formal property checking tools also present a challenge. While the current tools do provide an easier gateway to formal verification, there are still
areas of improvement that would help users accept the tools. These include increasing
the size of design that can be handled by the tool, more intuitive debug features, and
some method of code coverage. Users will be more likely to use the tool if it is more
intuitive and usable

Standardization of assertion libraries is a current concern in the rapid integration of
assertions. A full effort by users to integrate assertions into their design will not happen
until an accepted standard has been set. OVL is a currently available library written in
Verilog or VHDL so it can be used immediately in standard simulators, but is not
supported by all formal tools. Assertion standards currently being developed include
SystemVerilog Assertions and Sugar [Accellera]. The complete standardization and
acceptance of these libraries by all vendors will help in the acceptance of assertion-based
verification.

These are just some of the challenges that must be faced to allow formal assertion-
based verification to enter into the realm of standard verification methods. We’re
confident that with further development of formal tools and the continual integration of
these tools by more chip design groups, these challenges will be met and solved.

Conclusion

There are many advantages, but also many remaining challenges, involved in the
use of assertions and formal methods in the mainstream verification of an asic design. In
this paper, we have outlined techniques that were used to alleviate some of the problems
that were faced in introducing this new verification method. Constraint becomes the
largest roadblock when first verifying a particular design, but in using a hierarchical
approach we alleviated over-constraint while defining higher-level properties at the same
time. Particular pieces of shared code are best verified as a single entity, allowing for the
complete and exhaustive verification of its capabilities. With the use of these and other
techniques, assertion-based formal methods can become a mainstream approach to design
verification.

Contributors
Prashant Shah, Latha Ravindran, Barb Zivkov, Darryl Phillips, Swapnajit Mittra, Michael
Woodacre, Harry Foster

References

   Design: A Functional Coding Style Supporting Verifiable Processes in Verilog,
2. [Foster and Coelho] H. Foster, C. Coelho Jr., Assertions Targeting A Diverse Set
   of Verification Tools, HDLCon 2001