Streamline Verification Process with Formal Property Verification to Meet Highly Compressed Design Cycle

Prosenjit Chatterjee
NVIDIA Corporation
2701 San Thomas Expressway
Santa Clara, California 95056

ABSTRACT

In this paper, I describe a methodology and tool flow for using formal verification effectively to reduce the verification burden in large custom ASIC designs.

Categories and Subject Descriptors
B.8.1 [**Performance and Reliability**]: Reliability, Testing, and Fault-Tolerance.

General Terms
Algorithms, Reliability, Verification.

Keywords
Formal verification

1. INTRODUCTION

At NVIDIA, we design leading-edge, large graphics processor chips [1]. In order to achieve the highest quality and meet the market schedules, we employ and devise advanced tools and methodologies to meet our functional verification needs [2].

We have implemented a mixed verification flow consisting of multiple verification technologies. Formal verification is a key component in reducing the verification effort required to meet difficult compressed design cycle times. Combining verification technologies significantly reduces the overall design cycle while achieving greater confidence in the verification results. This methodology has been implemented and used effectively at NVIDIA in a production environment. This has allowed us not only to reach our conventional verification goals faster, but also to be able to achieve new goals in block-level verification previously not possible.

In Section 2, we describe the tools available to NVIDIA to devise our formal verification flow. Section 3 describes how we have used these tools to create a formal verification methodology flow that enables us to effectively leverage formal for different levels of user experience as well as different kinds of designs. Section 4 introduces the concept of planning the application of formal to different functionalities. In Sections 5, 6 and 7, we detail the methodology for internal property checking, coverage-targeted formal assisted simulation, and end-to-end data transport property checking, respectively (introduced in Section 3).

2. FORMAL TOOL TECHNOLOGIES

We have the following technology requirements as the basis of the formal verification methodology flow at NVIDIA:

1. *Constraint-driven simulation*: we use a simulator that can efficiently generate test vectors, frequently directed by user-provided input constraints and input biases. [1]
2. *Coverage measurement*: we use a tool that reports coverage results for functional, line, expression and FSM coverage. The tool also needs to be able to give access to uncovered targets, for use by our internally developed tools. [2]
3. *Semi-formal verification*: we use a semi-formal tool that is integrated into the simulator. The tool may also be able to statically prove some properties, and when the property complexity exceeds the tool capacity, the tool might give bounded proof results. This tool should also be able to cover user-specified coverage targets. [3]
4. *Pure formal verification*: we use a pure static formal tool to exhaustively prove internal properties as well as end-to-end data transport properties. The tool should be able to prove as many of these as possible automatically, and be able to shift into an interactive mode to reliably prove the larger end-to-end properties. Pure formal verification is necessary in regression since the analysis results are repeatable. [4]

3. NVIDIA'S FORMAL METHODOLOGY

We have built a flow that allows designers to adopt the benefits of formal with minimal verification expertise and learning curve, and still be able to prove the most complex data transport properties. Each verification technology, described earlier, needs to be matched to where it can provide the greatest value. Our methodology consists of the following components:

Internal Property Verification – We encourage designers to sprinkle internal implementation-level properties and assumptions while they are authoring the RTL. While these local properties do not replace block-level verification, they are useful in finding early bugs as well as assisting simulation by creating additional

[1] We use Synopsys VCS™ as our main simulator.

[2] This capability is provided within VCS.

[3] We use Synopsys Magellan™ for semi-formal verification tool.

[4] We use Jasper's JasperGold™ as our pure formal tool.

coverage targets. Internally at NVIDIA, our group has also developed a rich assertion library with assertion guidelines. Simulation, semi-formal and pure formal tools are useful in verifying internal properties. We discuss this in more detail in Section 5.

Formal Assisted Simulation – We use coverage goals to measure the effectiveness of simulation. Our coverage goals are internal (line, condition and user-specified) as well as functional (based on a testplan). The formal assisted simulation methodology uses formal verification to enhance the coverage achieved with traditional simulation methods. With an internal investment in flow automation, the technology is incremental to existing simulation flows and is fairly easy to adopt. The focus is on enhancing design coverage by targeting uncovered areas of the design through formal methods to generate additional simulation vectors. The design state space coverage is increased through formal methods. Effectiveness is measured by incremental coverage increase obtained over simulation alone. We have internally developed a complex verification infrastructure using the simulation, coverage and semi-formal tools to build this flow. We describe this in more detail in Section 6.

End-to-end Data Transport Verification – In order to get complete verification coverage for data transport blocks deemed as most critical for our application, we have to rely on proving a set of end-to-end data transport properties, that cover the functional correctness of the block. Internal properties or bounded proofs are useful sometimes, but are not sufficient to guarantee complete correctness of such properties. We have sometimes seen design bugs hidden by bounded proofs. Unbounded proofs ensure that a block adheres to its micro-architecture specification. They also can be run prior to when the testbench is available. We rely on a pure formal tool that gives us the flexibility to do this. This is described in Section 7.

To illustrate, the flow we use at NVIDIA is shown in Figure 1.

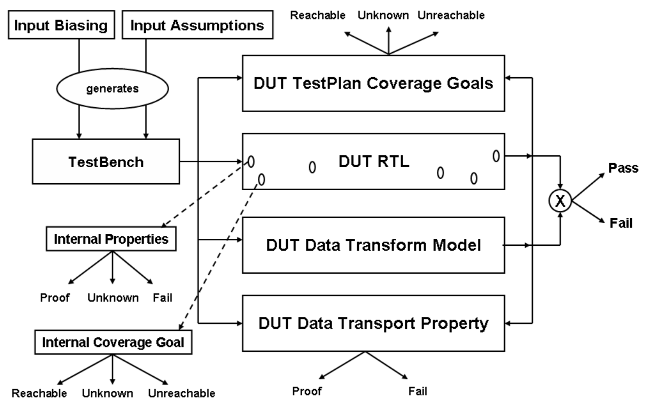

Figure 1. Formal methodology flow

4. FORMAL TESTPLANNING

Careful upfront planning is required to maximize the effectiveness of formal verification in shrinking the verification cycle time. Each verification technology needs to be matched to where it can provide the greatest value. NVIDIA uses multiple verification technologies in its verification testplan. A verification plan should carefully examine each of the blocks for formal suitability and rank the criticality of the blocks to overall functional correctness of the design. Critical blocks will be targeted for end-to-end verification to ensure block correctness. Blocks that are important but will be difficult to cover sufficiently using directed and random simulation should use formal assisted simulation to increase the coverage and improve bug detection. These have detailed coverage plans focusing on functional, line and expression coverage integrated into the testbench environment and automated to the fullest extent possible. Additionally, the use of internal properties provides additive value in increasing verification effectiveness.

Most data manipulation functionality falls into one of two categories; data transform or data transport control.

4.1 Data Transform

Data transform refers to mathematical manipulation of the values of data. Due to the complex mathematical manipulations, data transforms are best verified using a two model approach and vector driven coverage metrics. Starting with directed and constrained random simulation, we achieve initial coverage measurements. Formal assisted simulation is then used in to target design areas with low coverage in order to generate additional simulation vectors. We have applied the above technique on five such units.

4.2 Data Transport Control

Data transport control involves the transportation of data between processing units within a design. Data transport is typically not as concerned with the individual values of the data as with its uncorrupted transfer between design blocks. Control logic is an ideal candidate for automated model checking or formal verification methods, and exhaustive proof of proper behavior is a very achievable goal. High-level requirements (properties with high enough abstraction to be independent of the implemented logic) encompassing end-to-end functionality are generated and exhaustively proven using pure formal verification. These two data verification flows are shown in Figure 2. We have applied the above technique on twelve such units.

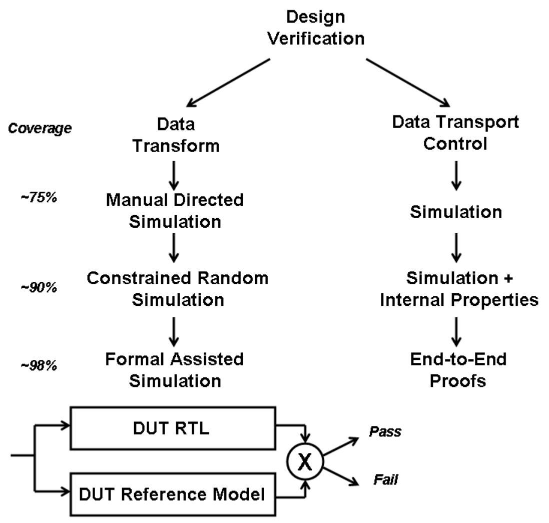

Figure 2. Data verification flow

5. INTERNAL PROPERTY CHECKING

We have adopted the use of formal assertion-based verification. This has enabled designers to build early confidence in the designs they develop. To facilitate easier adoption, we have internally developed a rich assertion library with assertion guidelines. Assertions that use this library are checked using simulation as well as by semi-formal and pure formal tools. In addition, many frequently used modules come with other internal assumptions and properties. These provide an easy introduction to formal for new users. Many designers also hand-code complex temporal assertions into their RTL. This is shown in Figure 3.

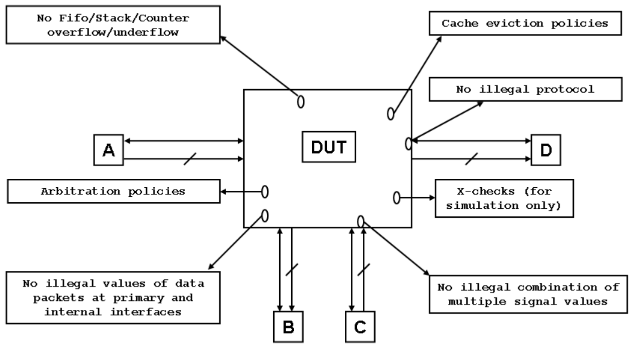
No Fifo/Stack/Counter overflow/underflow
Cache eviction policies
No illegal protocol
A
DUT
D
Arbitration policies
X-checks (for simulation only)
No illegal values of data packets at primary and internal interfaces
B
C
No illegal combination of multiple signal values

Figure 3. Internal property specification

Internal assumptions are important so that the formal tools do not give false negatives. NVIDIA project-specific internal standard properties are available in the assertion library. These are easily instantiated as assumptions.

Internal properties are verified formally with automated proof engines available from the semi-formal and pure formal tools. Frequently, a substantial fraction of these can be proved automatically. For the remaining, a bounded proof can give some confidence in the correctness of the properties. However, the bounded proof is not a complete analysis of the state space. In the past, we have experienced issues in sub-units where our semi-formal tools missed a deep bug that full formal analysis uncovered later. For this reason, on the most critical properties, it is desirable to get a full proof using a pure formal tool.

6. FORMAL ASSISTED SIMULATION

Our metrics to determine the effectiveness of simulation consist of both internal coverage goals and high-level testplan coverage goals. The internal goals are automatic (line and condition coverage) as well as user-specified (state transitions, interacting state transitions, internal stall conditions, data transform operands, etc). The testplan coverage goals encompass the design's high-level functional coverage goals. These cover different input packet types and sub-types, sequences of transactions, and checking under all modes of operation.

The standard method of verifying the data transform blocks using simulation is to compare the RTL against a functional data transform model as shown earlier in Figure 2. The input sequences are file-based and the file-based output sequences are "smart diffed". Input biasing provides a starting point for simulation.

Formal assisted simulation provides a means of reaching higher levels of design coverage through the use of bounded proofs driven from simulation stimulus [3]. For this analysis to be effective, properly constrained input vectors must be fed to the formal analysis as valid traces so that further bounded proofs can occur.

In the event a user has only simple input constraints, these can be added directly to the inputs of the design as shown in Figure 4a. Design constraints are written by the user and are then fed into the constraint solver in the simulator. The user can also add biasing to inputs. If the constraints are more complicated or involve a great deal of interaction with one another, a more custom approach is needed to avoid complexity for the constraint solver in the simulator. In this approach the user is required to write RTL code to implement the constraints as a generator (see Figure 4b). The constraint solver produces a testbench that only generates legal stimulus obeying the inputs constraints. This generated testbench is now a common environment for both biased random simulation, and formal verification reducing duplicated setup effort.

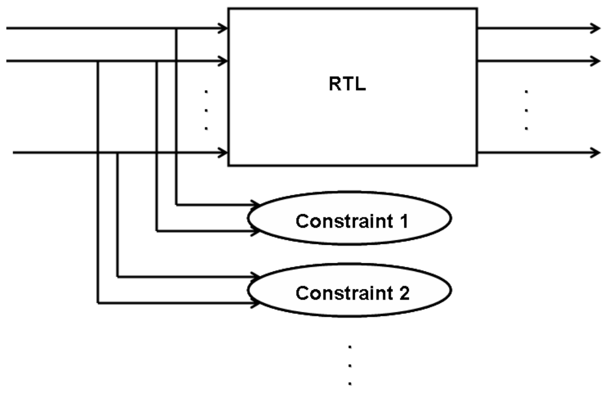

Figure 4a. Constraints written as assertions

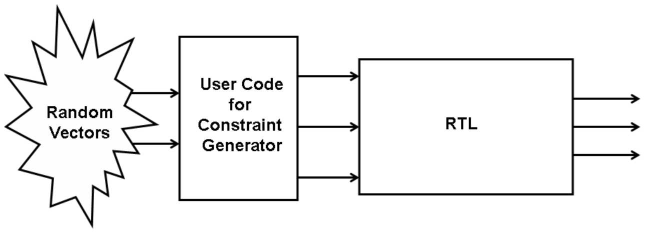

Figure 4b. Constraints implemented as a generator

At NVIDIA, we wanted to maximize the automation of this procedure in order for it to be widely usable by the designers. Prior to starting the current design project, we invested heavily in the generation of an internal environment to drive the formal assisted simulation methodology. This effort involved a number of design and tools experts working together to generate a truly customizable verification front end. We use the report files generated by the coverage measurement tools, and have built scripts to process these reports and farm off partitioned problems to semi-formal and pure formal tools. The individual test vectors generated by the semi-formal tools are stitched together by our internal scripts to create the desired input stimulus. Thanks to the efforts of the internal development group, the generator has allowed greater adoption of formal assisted simulation by the individual designers.

Formal assisted simulation works on the concept of the bounded proof. Simulation vectors from the constrained generator lead the analysis engines to “an interesting state” for further formal analysis [4]. An “interesting state” can be defined by user using assertion library or is automatically qualified by the semi-formal tool’s heuristics. This increases both coverage and consequently, bug detection. The bounded proof increases the depth of the state space search of the vector set with incremental extra effort on the part of the user. The downside of this is that it is not complete, and bugs can be missed due to bounded proofs. The user must be aware that the possibility still exists for bug escapes when using this method.

7. END-TO-END VERIFICATION

Pure formal verification is the only way to get full design confidence in critical blocks. Full formal proofs allow the absence of bugs to be confirmed in a design. Using high-level requirements for properties covers a large portion of the design space and ensures proper functionality at the microarchitecture level. Since it does not require a testbench to run, it can be used very early in the design flow where the cost of fixing a bug is low. The input constraints used for full formal proofs can be reused in semi-formal verification and vice-versa.

End-to-end verification involves creating a plain English verification testplan describing the list of micro-architecture requirements for a block. This testplan is analogous to a simulation testplan one might create if one were to verify a block using simulation. The important difference is that with simulation, even if all the requirement tests in the testplan are implemented, there is no guarantee that all input scenarios are covered. On the other hand, proving these high-level properties using a pure formal tool ensures complete coverage. The high-level properties are modeled and implemented using Verilog. In contrast with the internal properties (described in Section 5), the cone of influence for these high-level properties usually consists of almost the entire logic in the block; this tests the capacity limits of automatic formal tools.

When left on their own, many formal tools will attempt to read in all of the input logic in the cone of influence. This will create an intractable problem due the excessive size of the fanin logic. This is similar to the behavior of what a place and route tool would encounter if an entire chip were attempted to be routed all at once without any floorplanning. To deal with this, some tools employ automatic logic reduction using sophisticated heuristics (refer to the “Automatic region” in Figure 5). However, even though the problem size may be reduced, it may still be too big to solve by the formal engines, particularly for complex end-to-end properties. To make the problem tractable we must use a tool that allows an interactive use model that partitions the design complexity, and determines the optimal “Analysis region” (Figure 5). The analogy to floorplanning in formal verification is called “design tunneling” [5].

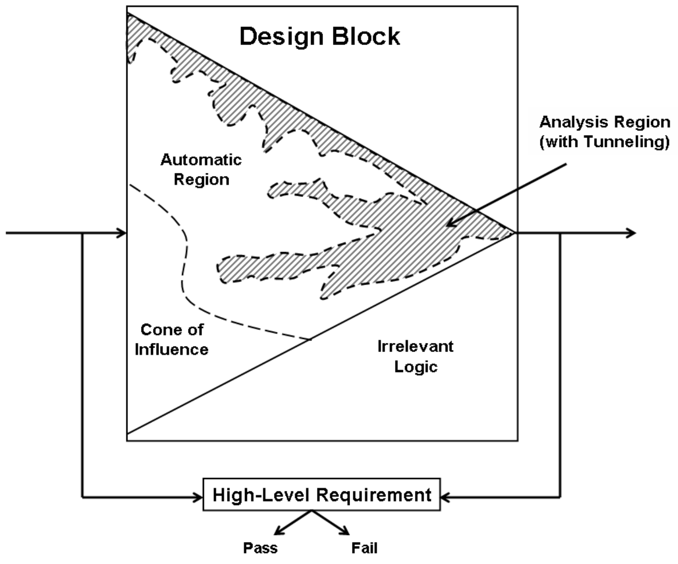

Figure 5. Design tunneling

Design tunneling enables the user to solve previously intractable problems by enabling the pure formal tool to consider only the logic which is relevant to the problem. This is automatically inferred from the user through an interactive debugging scenario [6]. By allowing the user to steer the formal analysis, the fanin cone can be selectively included to the point at which the problem becomes tractable, and exhaustive formal proofs can consequently be run on much larger designs than was previously possible.

8. CONCLUSION

Intelligent use of formal verification can provide many benefits towards shortening the verification cycle of a shorted design schedule. By properly targeting appropriate design logic, it is possible to achieve higher coverage in some cases, and exhaustive confidence in others. With the additional confidence it brings in the verification flow, designers can have higher confidence in their logic and managers can reduce the amount of chip respins due to undetected design problems showing up at the most inopportune moments in the final days before tapeout.

9. REFERENCES

[1] Malachowsky, C., "When 10M Gates Just Isn't Enough: The GPU Challenge", *DAC*, 2002.

[2] Smith, D., "NVIDIA: Scaling metholodogy", *Proceedings of EDP*, 2002.

[3] Magellan product description web site, http://www.synopsys.com/products/magellan/magellan.html, 2005.

[4] Ibid.

[5] Ip, N., and Foster, H., “Design Illumination”. *DesignCon 2005.*

[6] Jasper Design Automation, “JasperGold 3.1 Reference Manual”, *http://www.jasper-da.com/,* 2005.