

# Utilizing Assertion Synthesis to Achieve an Automated Assertion-Based Verification Methodology for Complex Graphics Chip Designs

Prosenjit Chatterjee

Saad Godil

Peter Nelson

Yuan Lu

Nvidia Corp.  
Santa Clara, CA  
pchatterjee@nvidia.com

Nvidia Corp.  
Santa Clara, CA  
sgodil@nvidia.com

Nvidia Corp.  
Santa Clara, CA  
pnelson@nvidia.com

NextOp Software, Inc.  
Santa Clara, CA  
yuan@nextopsoftware.com

**ABSTRACT** This paper reports a new functional verification methodology using assertion synthesis to automatically generate assertions and functional coverage goals. This new methodology complements and enhances our existing verification flows based on constrained random simulation, formal verification as well as emulation. Our experimental results supporting the new methodology are provided.

## Introduction

Each new generation of our state of the art graphics design enables more features and higher throughput. While some blocks are reused, key components typically needs to be re-architected and re-designed to accommodate the new features and to achieve higher performance. The new components and their interactions with the rest of the system create a daunting task for functional verification. With gate count exceeding hundreds of millions, black-box verification methodology alone no longer offers sufficient observability and scalability. Assertion-based verification addresses both issues by automatically embedding whitebox assertions and functional coverage goals as part of the verification process.

All of our in-house verification tools now support standard assertion languages such as SystemVerilog Assertion (SVA) [1]. Our main barrier to assertion-based verification proliferation has been the high effort required to create enough high quality assertions and functional coverage goals. It is desirable to have one assertion per 10 to 100 lines of RTL, but it is difficult to achieve this desired assertion density without over-burdening RTL designers [2].

An assertion synthesis tool named BugScope by NextOp is used for automatically generating assertions and functional coverage goals [3]. It takes the RTL design and its tests as input, and generates properties in SVA/PSL formats as output. The algorithm guarantees that the generated properties always hold true for the given set of tests. If a property is universally true, it can be classified as an assertion. Otherwise, the property must be an artifact of the tests and its negation represents a functional coverage hole.

In this study, we demonstrate a new and more effective coverage driven random simulation flow. We also show how to effectively integrate our multiple verification flows by using assertion synthesis technology.

## Assertion-based Verification

Assertions are now supported across all of our verification platforms including simulation, formal and emulation. The key challenge is the manual effort required to create an adequate number of high quality assertions and functional coverage goals. In practice, we see a disparity in both the quantity and quality of manually created properties among different design blocks even when writing assertions is a mandate.

We examined four exemplary modules from our GPU design. Module “ctrl” is the central dispatcher unit that handles multiple requests and decides how instructions are issued. Module “queue” is a queue with a complex scheduling mechanism. Module “entry” is a submodule of “queue” that maintains individual entries. Module “glue” is a crossbar logic for receiving and sending requests.

Data for the four modules are collected in Table 1. Column 3 shows the number of assertions manually written. Out of the four modules, only Glue contains assertions that are close to the suggested ratio of 1-10% of RTL lines.

Module Name	#RTL line	Manual	BugScope Assertion Synthesis		Assertions /RTL ratio
		#assertions	#reproduced	#total assertions	
Ctrl	22922	46	22	219	1%
Queue	3073	4	2	61	2%
Entry	4327	3	0	37	1%
Glue	38672	220	122	94	1%

Table 1. Property Statistics vs Manual Written Assertions

It takes one of our designers an average of one minute to classify a property. The number of assertions generated for each module is listed in Column 5. The synthesized assertions reproduced over 50% of manually created assertions. The number is listed in Column 4. Note that for the Glue module, synthesized assertions are more compact than manual ones. So even though there are only 94 synthesized assertions, they subsume 122 manual ones.

Table 2 lists some examples of assertions. The first two are manually written and reproduced by assertion synthesis. The last two are generated only by assertion synthesis. Note that

“@” stands for the temporal X operator which denotes the value of opcode at next clock cycle. Table 3 lists some of the functional coverage holes reported by assertion synthesis.

assert	sum(portX_dst_bitmap) <= 1, where X=0,1,3,5
assert	onehot0(valid_sp_sel)
assert	opcode == @opcode  > opcode != 3'h3
assert	replace[rptr_n]  > (age0 age1 arg2 age3) & 4'h8 != 4'h8

Table 2. Assertions Examples

cover	h1_su_valid && hX_su_valid, where X=6,8...16
cover	sum(hX_valid_spY) >40, where X = 2..7 and Y=0..7

Table 3. Functional Coverage Examples.

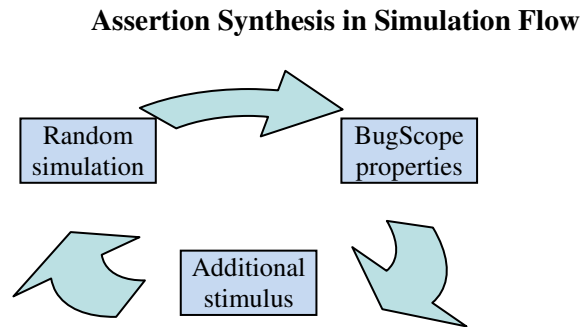


Figure 2. Diagram for Functional Coverage Driven Flow

Figure 2 illustrates how we integrated assertion synthesis in our functional coverage-driven simulation flow. We run assertion synthesis periodically throughout our verification process after significant changes in either the test or the RTL. The functional coverage goals drive further test development and the assertions ensure potential RTL problems are detected and localized for debugging.

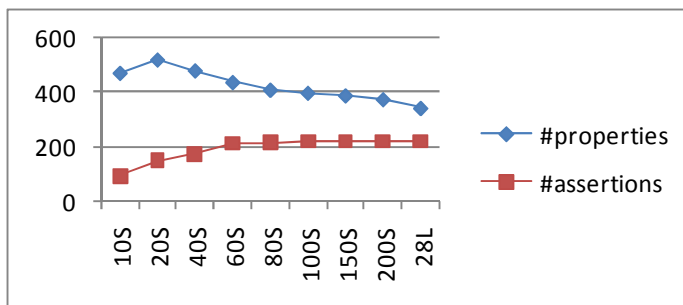


Figure 3. Relation between properties and given tests

Figure 3 shows the result of an experiment running assertion synthesis with increasingly more tests. As the number of the tests increases, the ratio of assertions over all properties increases as the coverage holes being patched by the additional tests. Eventually, the properties stabilize as additional tests no longer exercise new behaviors. We also notice that most assertions are created with a relatively small number of tests. Based on our experience, we propose the following methodology:

1. Early verification cycle, focus on
  - o collecting assertions
  - o addressing feature coverage issues
2. Mid verification cycle, focus on

- o Patching functional coverage holes
  - o Using assertions to find bugs and debug
3. Late verification cycle, focus on
    - o Corner case functional coverage holes
    - o Leveraging assertions/functional coverage in formal or emulation
  4. RTL release/tapeout, focus on
    - o Assertions as spec for future IP reuse

### Assertion Synthesis in Formal Flows

It is easy to recognize that the generated assertions can be used as targets for the formal verification tool to prove or disprove detecting corner case problems. But the coverage properties from assertion synthesis can also be put into good use in a formal flow. During formal verification, one can give both coverage properties and assertions as inputs to a formal tool. If a coverage property proves to be unreachable, it likely indicates an over-constrained formal environment. If the coverage property is reached by a formal tool, the formal engine must have found a trace that has not been exercised by simulation. As such, the coverage property can be an effective way to guide formal and semi-formal tool during the search for a counterexample of an assertion.

We applied assertion synthesis in combination with a semi-formal tool to verify a block in our GPU design. The block is a parameterized FIFO design with sophisticated credit control mechanism. The design had about 10K lines of code and assertion synthesis generated 83 properties including 45 coverage properties and 38 assertions. The formal tool was able to converge for 24 coverage and 21 assertions within 48 hours. All 21 assertions converged within 5 hours while some new coverage goals were hit after 40 hours. Note that the end-to-end checker for the FIFO design did not converge without manual abstraction of the memory elements. While patching the reported cover properties, a bug is found by the end-to-end checker and subsequently fixed using the formal flow with synthesized properties.

### Conclusion

An automated Assertion-based verification methodology is described in the paper. The new methodology allows us to uncover corner case bugs and identify functional coverage holes. By reusing the assertions and coverage holes across multiple verification flows, the methodology allows us to measure and leverage the quality of different test environments. Such an integrated verification platform is critical for verifying complex SoCs.

### Acknowledgement

We would like to thank all our colleagues for their help on the projects described in this paper, particularly, Marcio Oliveira and Xiaogang Qiu.

### References

[1] IEEE *Std 1800-2005*, IEEE Computer Society, 2005  
 [2] H. Foster, A. Krolnik, and D. Lacey, *Assertion-Based Design, 2nd ed.* Kluwer Academic Publishers, 2004.  
 [3] Nextop Software Inc. <http://www.nextopsoftware.com>