Reusing Testbench Components in a Hybrid Simulation-Formal Environment

Ritero Chi
Entropic Communications, Inc.
2570 North First Street
San Jose, California, U.S.A.
+1-408-325-8741
Ritero.Chi@entropic.com

Xiaolin Chen
Synopsys, Inc.
700 East Middlefield Road
Mountain View, California, U.S.A.
+1-650-584-4922
Xiaolin@synopsys.com

ABSTRACT

Simulation and formal verification traditionally have been treated as completely separate processes. Simulation is procedural and dynamic in nature, highly efficient at testing basic functionalities, but can be difficult to control to target corner case scenarios. Formal is static in nature and highly efficient at finding corner case bugs, but it has serious capacity limitation due to state explosion. Each has its own advantages and limitations.

Assertions used in formal can be reused and double checked in simulation, but reuse of non-formal-compatible testbench components by formal engines is not really possible. The reason is that formal algorithms require logic to be represented as a formal model. Testbench components are excluded from the pure formal verification environment because they cannot be compiled into a formal model. As assertion based verification methodology becomes more widely adopted, and formal verification increasingly becomes part of the verification flow, the need to bridge the gap between simulation and formal verification is also increasing.

Hybrid-formal technology can address this need. Hybrid-formal uses constraint-solving technology to generate legal stimuli compliant to the assertion constraints on the input ports. These constrained-random stimuli are driven to the DUT through the built-in simulator. Constrained random simulation is run in conjunction with formal, allowing formal search to be performed from deep design states, exploring corner case scenarios that are beyond the reach of pure formal engines and difficult to reach using simulation. Hybrid-formal technology combines the strengths from both verification methodologies to overcome the limitations of each individual technology. Because a logic simulator is one of the components in hybrid-formal, the technology allows the reuse of the testbench components in the constrained random simulation environment. Suddenly the door is open to more opportunities! The benefits include the possibility to:

- Identify discrepancies between a reference model and the DUT behavior in formal environment.
- Check for under-constraining, over-constraining, or conflict constraints in the formal environment.
- Monitor data transfer behavior and packet transactions, traditionally difficult to verify in pure formal environments

In this paper, methodology for testbench component reuse in hybrid-formal will be discussed, including the process of architecting assertions and testbenches for reuse at the verification planning stage. We will share the lessons learned during our quest for an ideal verification environment to face the challenges of finding corner case bugs in data-intensive designs and to reach the "last mile" of coverage convergence. We will describe the process we went through to overcome the hurdles encountered while integrating testbench components into the hybrid formal environment, which makes the ideal verification environment close to reality. Initial experiments at Entropic showed promise that adding testbench components to the power of formal could help reach the last bit of functional coverage convergence, verify functionality of data-path components, monitor data integrity and find corner case bugs. We see this has the potential to be a perfect vehicle to complement simulation and pure formal technology to ensure a high quality product!

1. INTRODUCTION

Assertion-based verification (ABV) has become a widely used methodology to address the challenges with traditional functional verification flow[1]. Formal verification is also gaining momentum. We started to explore new possibilities and wanted to see for ourselves what benefits the new methodology and technology could bring to the simulation testbench verification currently in place. We looked at what areas we would like to see improved in the existing flow, a few were easily identified: Verify legacy blocks with little or no documentation but to be used in next generation chips; find corner case bugs; improve code coverage convergence.

In the following sections, we will take a look at the the gap between simulation and formal. We will describe the challenges we encountered at different stages of adopting ABV methodology and formal technology, including creating and debugging assertions, verifying data intensive designs using formal, and attempting to improve coverage convergence with formal. We will share the lessons learned during our quest for an ideal verification environment and take you through the process we went through to search for a flow that could work for us.

2. EXISTING TESTBENCH ENVIRONMENT

In the existing simulation testbench environment, a VMM based transaction generator is the stimulus generator that creates transaction level traffic and drives them into the DUT using a simulator. There is a SystemC reference model. There are also VMM response checkers that check for the DUT response to the stimuli, and monitor how well the DUT has been exercised. As the simulation runs, responses from the DUT are compared to the expected responses from the SystemC reference model to check if the DUT is behaving correctly according to the reference model. At the same time, the VMM scoreboard information is also collected to measure the quality of the stimuli. A block diagram of the existing testbench environment is shown in Figure 1.

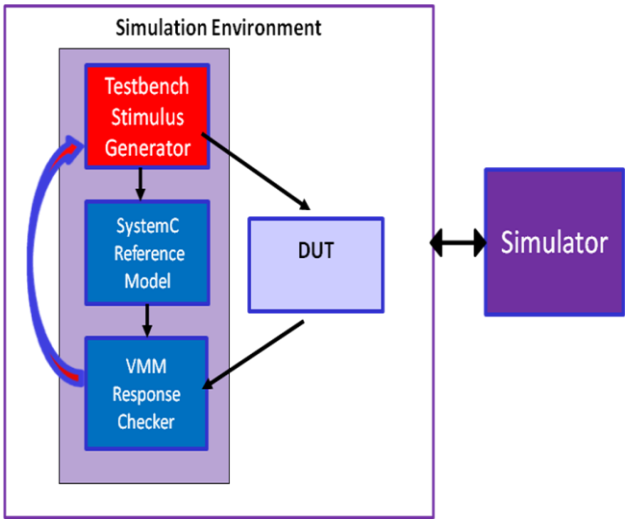

Figure 1: Existing testbench environment diagram

3. VENTURING WITH FORMAL

3.1 A Typical Formal Environment

When we started looking into formal verification, we learned that we needed to first have formal-friendly assertions, starting with the boundaries of the DUT. In a typical formal verification environment, the essential element is assertions. Given a complete set of assertions describing the protocol at the interface, many different formal algorithms can be applied to prove the assertions on the outputs or on internal logic to always hold true based on a set of assertions as constraints on the inputs. If assertions are violated, counter examples are generated to show assertion failures. Figure 2 shows the block diagram of a typical formal environment.

3.2 Debug Assertions in Familiar Environment

The first challenge we encountered was to get design and verification engineers to write and debug assertions for their own blocks. It was easy to get them trained in writing assertions in SystemVerilog language. However, it could be very time consuming to write and debug a set of complete interface assertions to describe interface protocol for the block under verification. Although formal checkers could provide an easy way to visualize and debug assertions without having to create a testbench, learning a new tool and debugging in an unfamiliar formal environment still made people shy away from engaging in ABV.

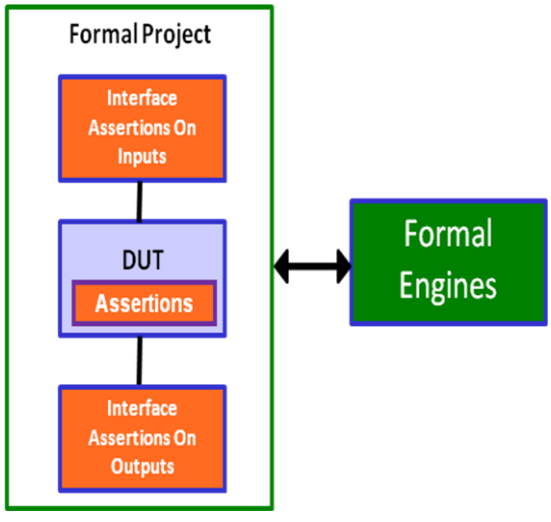

Figure 2: Typical formal environment

3.3 How to Make It Easier to Adopt ABV

To overcome this hurdle, two processes were put in place. The first was to identify a common set of protocols. This was easy to do since the protocol was the same for several RTL design blocks. A set of assertions describing the protocol was developed by the verification team and checked into an internal assertion library. The second process was to create a makefile that designers could use to visualize and debug custom assertions they wrote to verify their block. Since we used Magellan, a hybrid-formal tool with a built-in simulator, this makefile would automatically generate scripts to set up and run Magellan and bring up waveforms in the simulation debugger. This helped to ease the designer into the assertion world by providing a familiar debugging environment without a testbench. An example of the makefile is shown in Figure 3. Designer could simply type in "make debug_io", and out came the simulation waveform for assertion visualization and debugging. Figure 4 shows an illustrative waveform of constrained random simulation in DVE.

```
##########################################################################
#This rule creates a magellan session tcl file
##########################################################################
random_session:
    echo "read_project $(PROJ_NAME).prj" > mg_$(SESSION).tcl; \
    echo "run default –mode randomSim –maxTime $(RUN_TIME) –oneDebugFile" >>
mg_$(SESSION).tcl;

##########################################################################
#This rule runs the debug_io session
##########################################################################
debug_io: debug_io.filelist
    make SESSION=debug_io PROJ_NAME=$(PROJ_NAME)_io random_session; \
    mgsh -c -batch mg_debug_io.tcl -logPref mg_debug_io;
    dve –vpd mg_$SESSION/sim/magellan0.vpd
```

Figure 3: Example makefile

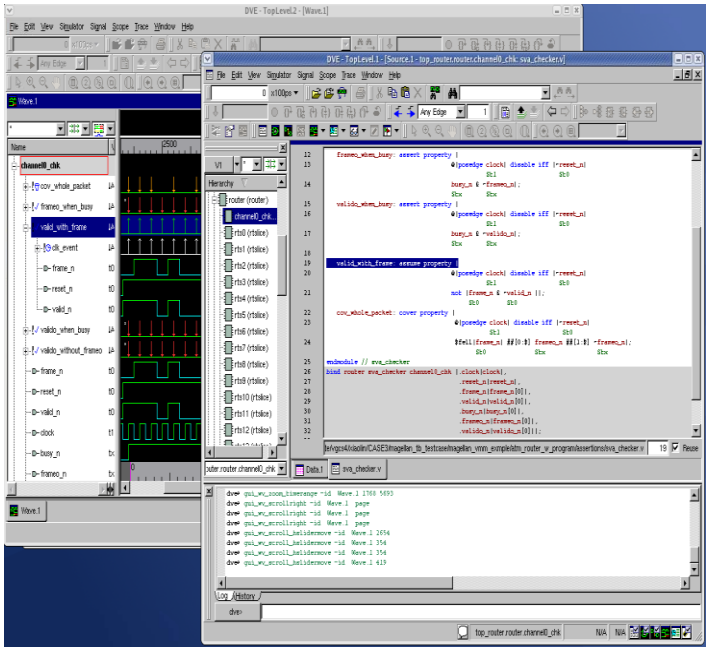

Figure 4: Waveform in DVE for assertion debugging

3.4 Running into Limitations with Formal

The next step was using Magellan to formally verify the assertion properties. We experimented with various sizes of design ranging from few thousand gates to the few hundreds of thousands to get a feel for what was possible or not with formal.

We were not able to get very far in terms of property convergence for very large scopes due to the data intensive nature of the designs.

For designs within 100k gates, and sequentials within 10% of the total gates, most of the automatic extracted properties and custom properties were validated. It turned out to be very promising in checking the behavior against interface protocol assertion properties in block-level designs. The tool could potentially be used for designers to validate their blocks at the end of their design phase before it is delivered for integration and verification in a higher level scope. Things seemed good as we would then be able to get the intent of the design and assumptions validated at block-level and have assertions be completely leveraged at the next level of integration.

However, there was still a missing piece in this flow: checking for data integrity. Due to the nature of the design being related to data processing, data integrity checking was an essential piece. We must have a way to check data flow in order to feel confident in this verification flow and quality of our design.

4. EXPLORING HYBRID-FORMAL

4.1 Leveraging Testbench Components in Hybrid-Formal Environment

The missing piece to make the flow really useful was to have the stimuli created during the formal session also being driven to the reference model so that we would have a way to make sure data integrity is not violated for the corner cases of the design.

Since Magellan used hybrid-formal technology, we considered the possibility of incorporating testbench components, especially the SystemC reference model and the cross-checking component, into the hybrid-formal environment. Referring back to Figure 1, many of the components, including some of the non-formal compatible ones, in fact can be leveraged[2]. This is because hybrid-formal technology uses a simulator in conjunction with formal engines.

We examined our existing simulation testbench setup. To begin with, neither the stimulus generator nor the feedback to the driver in the simulation testbench could be re-used. This was due to the fact that, in hybrid-formal environment, the testbench stimulus generator including the feedback mechanism in the simulation environment was replaced by a hybrid stimulus generator with constraint assertions on the inputs. These constraint assertions were solved by a built-in constraint solver and only the legal stimuli compliant to the constraints were driven to exercise the DUT through the simulator. The feedback driver could not be re-used because the stimulus generator was not accessible to user for adding the feedback driver to the stimulus generator.

Other components in our simulation testbench environment, even if non-formal compatible, such as the response checker, reference model, coverage and data collection component, passive feedback, and some of the assertions were using non-formal compatible 4-state logic could actually all be re-used in the hybrid-formal checker's simulation environment.

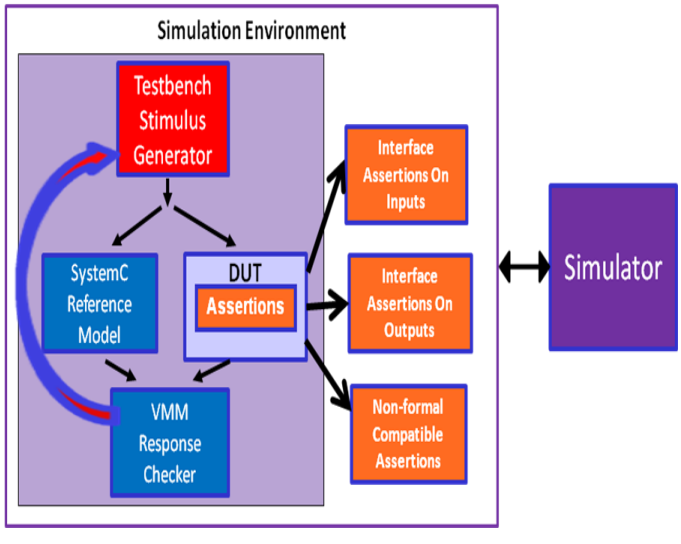

Figure 5: Assertion-based verification testbench environment with hybrid-formal reusable components

Once we established what components can be re-used in hybrid-formal, the next task was to go through the testbench setup and organize it so that the driving and active feedback mechanisms were completely separated from the monitoring and data collection components. This took some re-thinking and re-architecting because the testbench was already done in such a way that all the driving, feedback and monitoring functions were intertwined together in a transaction-level fashion. We had to make sure that the code only seen by the simulator in the hybrid-formal set up did not contain any logic feeding into the fan-in cone of an assertion targeted by formal engines or used as a constraint. Otherwise, the results could be inaccurate or mismatched. The interface to the components had to be at the signal level. Figure 5 shows the new simulation testbench architecture and Figure 6 shows the testbench components being re-used in hybrid-formal environment.

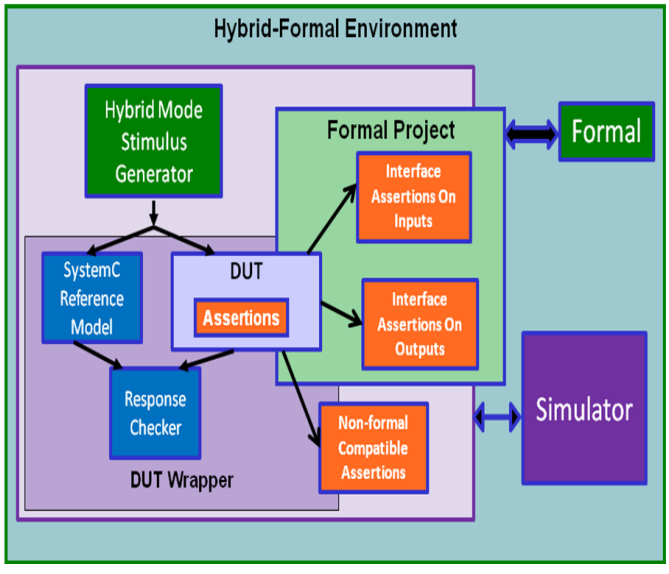

Figure 6: Leveraging testbench components in hybrid-formal

4.2 Improving Coverage Convergence with Hybrid-Formal Technology

Code coverage is one of the sanity measurements to gauge the quality of the simulation testbench as well as the RTL design code, and almost always used as one of the sign-off requirements. It can be really difficult to reach the final bits of coverage targets.

The good thing is that Magellan offers built-in automatically extracted properties for line and condition coverage. Magellan and VCS also shares the same coverage database. We could import the coverage database obtained from simulation testbench, and Magellan would automatically target only the coverage goals that were un-covered in the database. This feature could help improve code coverage convergence in two ways. Firstly, without any constraints, we found that we could easily identify the unreachable line and condition coverage goals. We could then exclude them from total number of goals in the testbench metric, therefore immediately improving the coverage results. Secondly, when we were certain that the set of constraints at the block interface level were adequately representing the set up of the simulation environment, coverage goals reported by hybrid-formal technology could easily be imported into testbench simulation coverage database, therefore increasing the coverage convergence outcome.

Similarly, hybrid-formal can also assist in improving coverage convergence in functional coverage. Although covergroups were not yet supported in Magellan's formal flow, they could easily be converted into cover properties which were supported. Usually the functional coverage points are intended to be checked by higher-level tests, and facilities like Coverage Convergence Technology (CCT) can help reduce the manual effort and the verification time to reach the verification goals[3], but CCT cannot determine if coverage targets are unreachable. Formal technology can be used at block level to check for unreachability of these functional coverage points without constraints or with proper constraints. This can help identify potential set up or design flaws preventing the coverage points to be unreachable. However, to consider functional coverage targets reached in the hybrid-formal environment, it is absolutely essential to make certain that the assumptions made at the block level are equivalent to the stimulus driven at the higher-level.

Although we did not have the time to reproduce test results on the entire block, some experiments on smaller blocks demonstrated the ease of integrating hybrid-formal technology into the coverage convergence flow, especially the unreachable coverage targets with unconstrained environment.

4.3 How to Incorporate Testbench Components

When incorporating testbench code into Magellan, we found there were two ways to make this happen: One way was to create a top level wrapper that instantiates the program block or testbench modules along with DUT. Another way was to instantiate the classes directly inlined in the Magellan hybrid-formal environment without the program block.

Again, we believed automation was the key to facilitate adoption. Makefiles were created script to do the following:

- Automatically generate top level wrapper that incorporated testbench components
- Compile the SystemC reference model for reuse in Magellan environment
- Automatically including assertions from the central library as well as custom assertions written by the designer based on the block information provided
- Set up and run Magellan with compiled library along with other testbench components including scoreboard and passive monitors in the simulation environment to target assertions and coverage goals

At the end of the run, property falsification traces were automatically displayed by the tool for designers to diagnose the cause of failure, coverage information could be displayed in URG, and scoreboard summary information could also be gathered from the final block just before exiting from the simulation in Magellan.

4.4 The Power of Hybrid-Formal Technology

Why should anybody get excited about being able to reuse testbench code in formal? What are the benefits you get out of this?

From our experiment with this setup, we found that the benefits included:

First, it helped to identify discrepancies between reference model and DUT without a full-fledge block level testbench, using just the assertions as constraints for Magellan's stimulus generation.

Second, the added monitors, both functional checkers as well as coverage monitors helped to identify conflict constraints, over-constraining, or under-constraining in the formal environment. These constraints were the very same ones used for proving properties. It was already known that assertions in formal should also be used in the testbench to check for discrepancies in the assumptions made in the formal environment. By having reference models and monitors in the hybrid-formal environment, it allowed us to also double check the correctness of assumptions made for formal environment without a testbench.

Third, this allowed the monitoring of data transfer behavior, such as packet transactions which is difficult to verify in pure formal environment due to capacity limitations.

Fourth, by adding a reference model and response checker, it provided a way to help with coverage convergence. With this setup, if the DUT behaved as expected from the reference model and none

of the checkers fired, the reachable coverage might be considered merging into the simulation testbench coverage matrix. Not to mention the unreachable coverage goals proven by formal engines without any constraints or with proper constraints could also be merged into simulation testbench coverage matrix.

5. LESSONS LEARNED

5. 1 Architecting Assertions for Formal

Plan for assertion writing from the beginning with formal verification in mind. First partition blocks so they have clear and complete interface protocol definition. Start assertion development early in the cycle, even before or in parallel with RTL and testbench development! Make sure that you have a complete and accurate set of assertions for the interface. Assertions scattered here and there are not going to be adequate for formal verification later. Standardizing the interface used by all the various blocks is key to guarantee the effort in coming up with the interface properties is reused and therefore making the process efficient.

Try to keep non-formal compatible assertions separate using macros to hide them from formal. Some formal unfriendly assertions are formal-compatible but produce too many sequential elements exceeding formal engine capacity limit. Put these formal-unfriendly assertions in macros as well to hide them from formal. These assertions are only seen by the simulator and can be checked in hybrid formal mode.

5.2 Architecting Testbench for Hybrid-Formal

In the beginning of the testbench planning stage, some thought should be put into architecting the testbench for later re-use in formal. Keep the stimulus generator part modular so they can be easily separated out later for hybrid-formal verification.

It is common practice to hook the layered adaptive transaction level stimulus generation directly to the monitor and driver. However, it will take a lot of work later to separate the scoreboard from the transaction layer for reuse in hybrid-formal environment. Keep this in mind when constructing monitors and feedback connections to the stimulus generator. Try to use lower level abstractions at the signal level for monitoring, coverage or data collection rather than at transaction level, so they can be later removed for hybrid-formal verification. For example, some good testbench practices for testbench reuse include:

- Use lower level monitor for scoreboard control, instead of at transaction level. Good and bad examples are illustrated in Figure 7 and Figure 8.
- Keep the consensus part of the environment completely separate from passive monitor.
- Some of the feedback constraints such as biasing of the input distribution could be used in hybrid environment, so make them as a separately contained component to be later re-used in hybrid-formal environment.
- Keep signal level constraints separate from high level transaction level constraints, so that the signal level constraints can be ported to hybrid-formal environment.
- Keep active response checkers and reference model separate from passive ones, so the passive monitors and reference models can easily be incorporated into the hybrid-formal environment later.

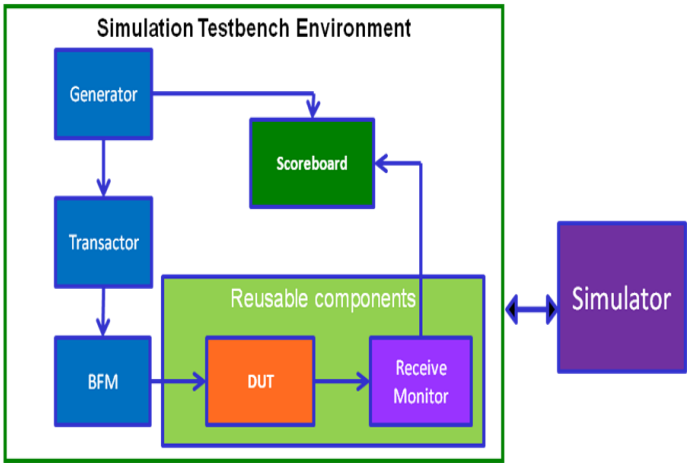

Figure 7: Transaction level scoreboard not reusable in hybrid-formal

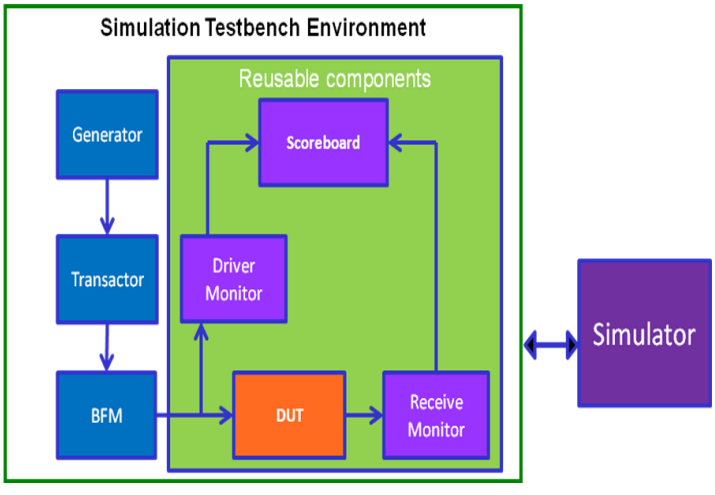

Figure 8: Signal-level scoreboard can be reused in hybrid-formal

5. 3 Establishing a Basic Flow

Defining a process is the key to deploying ABV methodology and formal technology into the existing verification flow. One example of a basic flow is shown in Figure 9.

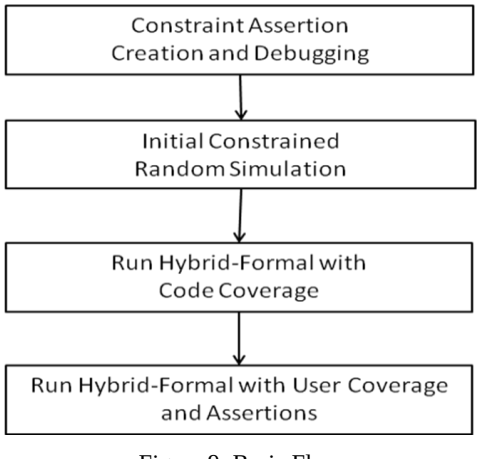

Figure 9: Basic Flow

5. 4 Automation Saves Time!

It is clear to us that developing the scripts to automate the steps in the process makes it much easier to adopt this verification flow. Creating the central library, defining procedures for developing custom assertions and testbench components allow the possibility for automation, and automation does increase productivity!

6. SUMMARY

The results of our experiments have showed promise that adding testbench components to the power of formal could help reach the last bit of functional coverage convergence, check data integrity and find corner case bugs. We believe this has the potential to be a perfect vehicle to bridge the gap between testbench simulation and pure formal technology.

7. ACKNOWLEDGMENTS

The authors wish to thank Wei-Hua Han for his contribution to the SystemC compilation, VMM and Magellan flow.

8. REFERENCES

[1] Lionel Bening and Harry Foster, "Principles of Verifiable RTL Design", ©2000, Kluwer Academic Publishers, pp. 31-32.

[2] Xiaolin Chen, Mandar Munishwar, Den Benua, Krishna Balachandran, "Combining Formal Verification with Simulation: The Best of Both Worlds ", Synopsys Webinar 2009.

[3] Simon Huang, Zhiyong Shao, "Improve Verification Productivity with Synopsys Coverage Convergence Technology", SNUG 2009.