Marrying Simulation and Formal Made Easier!

Lun Li, Durga Rangarajan, Christopher Starr, James Greene
Samsung Austin R&D Center (SARC), Austin, TX 78730
Nitin Mhaske
Synopsys Inc, Austin, TX 78746

***Abstract*- How formal and simulation can benefit each other to reduce the verification cycle is a very interesting topic. This paper details our experience when an assertion synthesis tool was used successfully to augment simulation and formal to improve productivity and quality. These properties from an assertion synthesis tool are reused automatically as SVA assertions in the formal environment. If any property is violated in formal, it could be a constraint issue, a coverage hole from simulation or a design bug. Properties proven passing in formal are then reused in IP and SoC simulations as golden assertions. This effectively reduces overall efforts on both simulation and formal and improves verification quality.**

I. Introduction

Often, simulation and formal are considered two different worlds; overlapping but without directly reducing efforts or improving the quality of each other. Technologies like hybrid formal, code coverage un-reachability using formal help to bridge them with some success. But developing a complete and accurate set of constraints for formal functional verification right from scratch is often error prone, iterative and a time consuming process. At the same time, formal efforts are not directly useful to find unknown simulation coverage holes; which is the main signoff flow. This paper will share how assertion synthesis technology by Synopsys's BugScope [1] was used in leading mobile processor design to augment simulation and formal verification and found constraint issues, coverage holes and bugs.

II. Previous work

The usage of formal tools in the verification of complicated designs has been growing by leaps and bounds in recent years. If a formal test bench has been constructed for a RTL block, it can be used to discover the root cause of a failure observed in post silicon testing. Often the stimulus needed to generate the failure can be developed in much less time than using traditional stimulation methods. Generating stimulus and bringing up a new design can also be done in less time using formal techniques [2].

Another important use of formal techniques is in situations where it would take too many simulation cycles to thoroughly verify a piece of logic. Some examples of this are floating point units and clock gating [3]. For these applications formal tools can test more thoroughly in a reasonable amount of time.

Properties developed for use in a formal environment can be used in simulation environments and vice versa. Assertions developed by RTL designers that never fail in simulation can be tested further by formal tools. This may uncover stimulus holes in simulation or badly written properties [4].

III. Challenges/Issues of Previous Work

For successful use of formal techniques, it is important to develop a correct and complete set of constraints. Developing constraints in a synthesizable, non-behavioral format from scratch is a time consuming and iterative process. An incorrectly constrained formal environment may yield a false positive result, hiding a bug, or a false negative, wasting too much time debugging the incorrect constraints.

Also for effective bug hunting with formal, it is important to have a good combination of end-to-end checker assertions as well as micro-architectural based assertions. End-to-end checkers capture the high level specification functionality and, when proven, gives significant confidence over simulation. They often have a larger cone of logic and suffer from convergence issues due to the capacity limitations of formal tools. Micro-architectural properties, although harder to map to the specification, are relatively easier to converge and enable effective bug hunting. However, writing a good set of micro-architectural properties involves significant amount of designers' time and expertise.

IV. Formal Related Work

Our design supports three legacy ARM protocols and three customized protocols. We use standard Asserted-Based Verification IP (ABVIP) from a third party to verify the three legacy ARM protocols. The challenge to the verification team was applying this verification method to the three customized protocols. We have two different approaches for the three customized protocols: one approach is to reuse the standard protocol ABVIPs; the second approach is to develop our own protocol ABVIP from scratch. Unlike simulation, where the unit and top level verification are built separately, in our formal environment, we build the unit/top level environment together to maximize reuse. TCL files are used to separate the unit-level/top-level setup through the black-boxing of unused modules and converting checkers into constraints, if needed. We also developed some utilities to speed up formal debug and result tracking.

A. Reusing Standard Protocol ABVIP

For interfaces that closely resemble industry-standard protocols, we used a series of gaskets to convert the standard protocols into the customized protocols used in our design. This greatly speeded up the development and deployment of the test bench.

Each gasket would instantiate a copy of the industry-standard protocol ABVIP with a top layer of assumptions/assertions that constrain the input stimulus generated by the standard protocol to match our customizations. For example, the custom protocol implemented by Samsung allows a different number of read and write requests to be issued in one cycle versus the industry standard protocol. This was implemented with an assumption that changes the allowed number of requests at the same time.

B. Developing our own Protocol ABVIP

One of our three customized protocols is a packet-based protocol that is used within our design. It is used throughout our formal environments. In the top-level, it is used as a monitor/checker that can capture failures before they propagate to an external interface. In unit level, it acts as a driver/constraint to generate legal inputs for the formal environment. The ABVIP was developed with a layered approach with each layer implementing a well-defined portion of the protocol. The first layer would only be concerned with properties that happen in a single cycle. For example, the format of a read or write request would be constrained by this layer. The single cycle layer would also generate signals to be used by the other layers to make the writing of assumptions and assertions faster and easier to understand. The upper layers would be presented with a simple derived signal indicating that a request was available instead of having to decode multiple signals.

The second ABVIP layer handles properties that involved complete transactions. Using signals generated by the single cycle layer, a read transaction could be generated or monitored throughout all the different phases of its life cycle. This layer would see a read request and then create the stimulus for a read response and a read acknowledge. This transaction layer would not contain properties that involved relationships between the different transactions like address hazards or other ordering rules.

The third ABVIP layer above the transaction layer handles properties that involve transaction relationships. For example, the standard protocol does not allow a read response to be sent if there is an outstanding snoop to the same address. The ABVIP layer generates signals that track hazards between the transactions to be used to create stimulus scenarios. If a formal test case was required to create a hazard condition between a snoop and a read response, the signals from this layer could be used to generate cover properties showing an example of the stimulus. Very often, it was much faster to generate these stimulus scenarios than with traditional simulation environments.

The final ABVIP layer handles properties that are unique to an interface because of where it resides in the topology. Some interfaces have different transaction types and ordering rules. For example, one particular interface may generate snoop requests based on a coherent read or write request. Another interface may not do this. Address hazarding rules may be customized for different interfaces. The rules of this layer allow the base ABVIP behavior to be tailored to each individual interface depending on its requirements and reused throughout the test benches.

C. Layered Approach to Create Unit and Top Level Formal Environment

Due to the capacity constraints of formal tools, it is difficult to have end-to-end properties that converge into a proof or a counter example in a reasonable amount of time at the top-level. This required the development of unit level test benches to allow deeper exploration of the design. The ABVIP described in previous section allows us to build unit-level environments.

Our formal environment was developed with a layered approach. The layers were designed to allow maximum reuse of SVA assumptions and assertions throughout the top-level and unit-level formal test benches. We build unit-

level/top-level formal environments together in one shot. TCL files are used to separate unit-level/top-level setup by black-boxing unused modules and converting checkers into constraints, if needed.

The formal gaskets we developed could act as an active component to drive stimulus into a unit-level test bench. They could also be configured to act only as a monitor to allow them to be included in the top-level test bench. The gaskets were also highly parameterized to allow them to adapt to behavioral differences between otherwise similar interfaces. For example, the format of the transaction identifiers would be different depending on where the interface was instantiated in the topology. This configurability of the gaskets greatly reduced the amount of formal code needed to support all of the unit-level and top-level test benches.

Another challenge for our formal verification environment was the need to support multiple versions of the test bench for the series of designs we were developing. This was handled by structuring the formal code into sections that were reusable for every design in the series and sections that were specific to a particular design. This allowed us to bring up a formal test bench for a new design in a matter of days instead of weeks.

D. Utilities to Speedup Debug and Report progress

To speed up the debugging of counter examples, we developed a TCL script to trace the activity of each interface type being used in a particular test bench. The trace logging allowed the stimulus activity to be display as a text file which was more useful than looking at waveforms. The text logs could be used when filing bug reports to clarify the issue being discussed. Trace logs from different interface types could also be interleaved on a cycle-by-cycle basis to make the ordering of transaction events easier to understand.

Other TCL based utilities included a script to convert the CEX and cover traces to Verdi compatible data bases because the RTL designers were more comfortable using that tool. Another script would restore a Jasper data base created from a nightly regression to allow a closer inspection of the CEXs. Scripts were also developed to find dead RTL code and unreachable coverage properties.

Scripts were also developed to take the results from the nightly regression runs and create a high-level view in the form of graphical charts for management to track our progress. These graphical views would keep a running tally of the number of completed proofs, the number of CEXs and the number of unreachable covers over the course of the project. This allowed our status to be presented in a way that was similar to conventional simulation.

V. ASSERTION SYNTHESIS RELATED WORK

The assertion synthesis tool is deployed in two generations of our design. In the first generation, the assertion synthesis tool was deployed in the post silicon stage with mature simulation test benches, high coverage and exhaustive daily regressions. Other higher level test benches were also running regressions in addition to the IP verification team. The assertion synthesis tool was deployed in four individual unit-level simulation test benches and the top-level simulation test bench. Each module was individually studied using the tool results in collaboration with formal. The tool has also been deployed in the next incremental project along with the updated simulation environment.

A. Assertion Synthesis Tool Deployment Flow

The first step is to generate properties for the RTL module by running unit level regressions with the assertion synthesis tool enabled. This will generate properties that always hold in the unit level simulations. The assertion synthesis tool we are using by default generates level_0 (one signal only), level_1 (2 signals related) and level_2 (more than 2 signal related) properties. The level_2 properties were found to have diminishing returns in terms of value of the property when taking into account the time and effort spent analyzing them. So they were ignored for further analysis.

The next step would be to classify the level_0 and level_1 properties as assertions or coverage properties. Properties that are true in the design should be classified as assertions. Properties that are coverage holes in simulation environment but not necessarily always true in design are classified as coverage properties. Instead of spending time and effort in classifying the properties, the properties were classified by default as assertion and reclassified appropriately later on. There will be a future reliance on the formal environment and the top level regressions to confirm the hypothesis that the unit level simulation environment is golden and generating only valid assertions. The properties generated by the assertion synthesis tool are converted to SVA's and are bound to the RTL module.

The SVA'S generated were then run in the top-level simulation environment. The assertion fails at this stage indicate that the classification of assertion was incorrect or the top level environment has illegal stimulus. If the classification is found incorrect on analysis, that indicates the top level had stimulus that the unit level was missing. If this was the case, the classification should be changed to coverage at this point. If the top level had illegal

stimulus, the assertion classification of the property holds and the simulation environment needs to change the stimulus. The SVA's generated after this stage have a higher confidence level because they were tested in two simulation environments. These SVA's could then be deployed in various other higher levels of hierarchy to further test our classification and gain confidence.

The SVA's are now ready to be run in the formal bench. The formal run would give us data on which assertion properties are proven, which are undetermined and which had counter examples. The proven properties at this point are golden. The properties for which a counter example (CEX) was found will have to be analyzed to see if the test benches have a stimulus hole or if the formal bench needs a new constraint that was missed. The counter example (CEX) has to be reclassified as cover property or the formal constraint needs to be fixed at this point. If the assertion was found to be a cover property, all the test benches analyzed so far will have to create stimulus to cover this scenario.

Once the counter example issues are resolved, the SVA's can be run again in the simulation environments and the formal bench. This process can be repeated till all the counter examples are resolved. A point of interest could be the convergence rate of properties; this could be an indicator for the formal test bench health and the design complexity.

The flow described above is summarized in Figure 1.

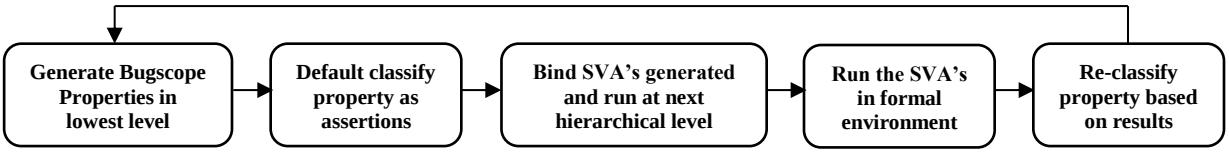

Figure 1. Assertion synthesis tool Flow

B. Assertion Synthesis properties

Properties are generated from the assertion synthesis tool for every unit based on unit level simulation regression tests. Mainly control intensive modules in every unit and the protocol ABVIP (active in simulation) were targeted to generate properties. Properties generated at the RTL modules are mainly white-box properties that capture the micro-architecture and implementation specific design intent.

Examples of such properties are:

- `credit_cnt <= 5'h1B //credit counter never exceeds 'h1B`
- `gray_code(SfSetCrqsMemWr)//SfSetCrqsMemWr always follow gray code, only one bit sets/resets`
- `overflow(PromoteTimer) |-> PromotTimerClr // PromoteTimer will never go from 'h1111 to 'h0000 except when PromotTimerClr is asserted`
- `countones(wack_buf_valid) < = 8'h5 // number of bits set in wack buff are less than 6`
- `mutex(NoWrLu0, NoWrLu1) //NoWrLu0 and NoWrLu1 are mutually exclusive, not set in the same cycle together`

Since we used a packet-based protocol in our design, the assertion synthesis tool generated many properties related to fields inside one packet. However, due to limitations of the simulation stimulus, not all the fields are equally exercised and many properties related to the packet field are incomplete. To improve the quality of the generated properties in this area, we imported our ABVIP developed in formal environment into simulation as well. With the targeted ABVIP modules, the quality of generated properties around interface improved significantly. Roughly 70-80% of the properties associated with the packet-based protocol were replaced using signals in ABVIP.

Examples of properties captured at ABVIP modules are:

- `rdy ==1'b1 // ready is always asserted, as stalling of particular interface is not supported`
- `mutex(MPACE_WLAST0_crq, MPACE_WLAST1_crq) // two last signals cannot assert same time`
- `Onehot0(FpaceCrdDestVld) // At-most one bit is set in the FpaceCrdDestVld signal`
- `AwReqS1Rdy |-> AwReqS1Vld // When ready is high, valid has to be high`
- `inside(pkt_len, 6'h0, 6'h3) // packet length is always 0 or 3`

VI. Combining Assertion synthesis and Formal Together

The assertion synthesis effort initially started as a separate effort to increase verification confidence, as well as the formal verification effort. However, the nature of properties allowed us to combine both of them together to benefit each other, especially during the process of developing our own ABVIP from scratch. In the following section, we describe how we combined the two methods into one flow.

A. Flow

Assertion synthesis technology generates properties that are TRUE by analyzing all of the simulation regression tests passing at the block level. These properties are reused automatically as SVA assertions in the formal environment. If any property is violated in formal, it could be a constraint issue, a coverage hole from simulation or a design bug. Properties that fire due to a constraint issue are used to correct the constraints or augment additional constraints, bootstrapping the development of formal constraints. Properties that fire due to simulation coverage holes help to identify missing stimulus or sequence issues. If a property is an assertion and fires under the correct constraints, it likely indicates a design bug. Properties proven to pass in formal are then reused back in IP and SoC simulations as golden assertions. Optionally, the properties found covered in top level regressions or deemed coverage holes upon manual review are ported to formal as cover properties and may expose additional constraint issues. This effectively reduces overall efforts on both simulation and formal and improves verification quality.

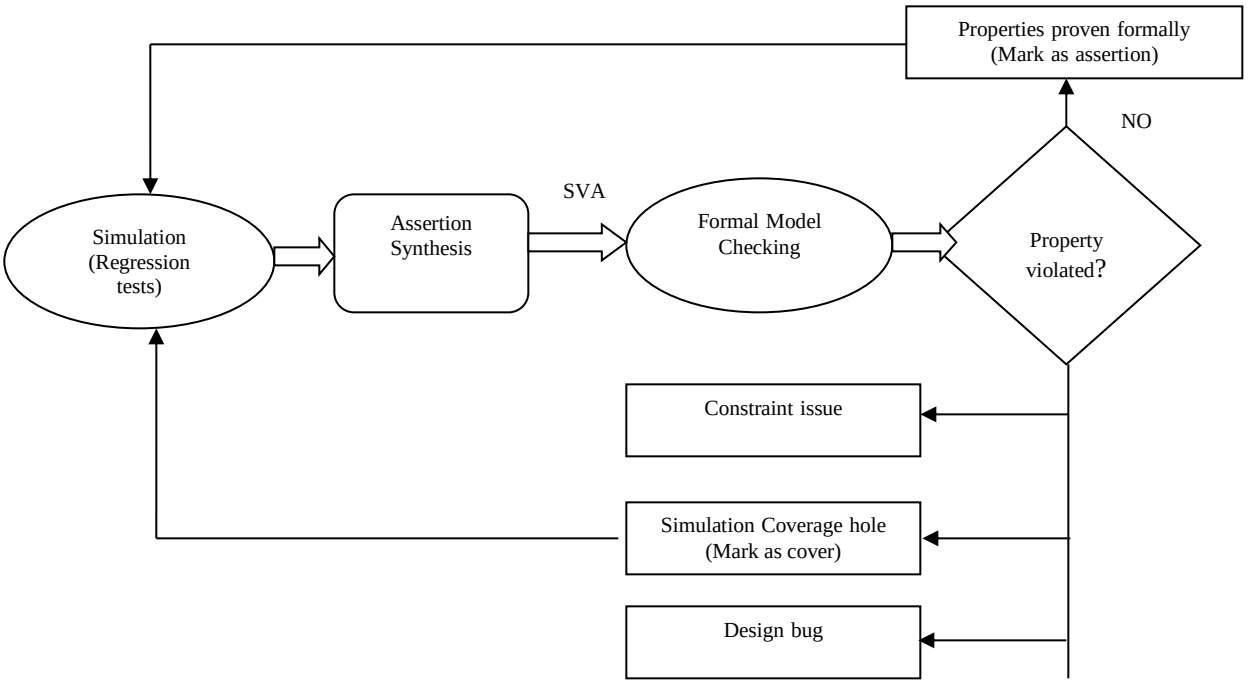

Figure 2. Assertion synthesis and Formal Combing Flow

B. Summarized Deviation From Typical Assertion Synthesis Flow

In a typical simulation-only flow, the process of classifying the properties as assertions or coverage properties could be labor intensive. To improve ROI, in conjunction with formal, the following deviations from a typical assertion synthesis flow were taken.

- Properties are generated when simulation is at a mature stage; so coverage holes and associated noisy properties are fewer.
- The level_2 properties (involving more than two signals or expressions) were found to have diminishing returns in terms of value of the property when taking into account the time and effort spent analyzing them. These properties were ignored.

- Instead of classifying properties, the properties were auto-classified without review as assertions first and taken through multiple stages of filtering. At each filtering stage, waveforms help to review the properties efficiently and help the respective environments to mature.
    - Stage 1: Assertions are run in multiple unit level random regressions to see if any additional coverage scenarios are hit. If any of assertions fire in random regressions, they are reviewed with waveforms. If it is a coverage scenario that was not hit in first set of regressions when properties are generated, but are hit in additional random regressions, they are marked as "coverage".
    - Stage 2: The remaining assertions were then run in the top level simulation environment and the violations are reviewed with waveforms. Violations would be fewer in the case of mature unit level simulation.
      If there are important unit-level coverage holes hit in the top level regressions, they are marked as "coverage" and tests are added to target them. If they are violated due to trivial differences between two simulation environments (unit versus top), that does not affect overall functionality; they are marked as "ignore" and are entirely omitted from further analysis. If the violations are due to top level simulation environment not behaving per specification, the assertion is fixed to correct it.
    - Stage 3: Un-violated properties from the unit and top level simulations are then in run in formal and violations from formal are reviewed with counter example waveforms. Violated properties could be constraint issues, coverage holes missed in the unit/top level simulations, or design bugs. Passing properties are then automatically marked as assertions. Inconclusive properties may need a manual review and this is done as needed mainly for highly critical modules.

C. Constraints in Formal

The assertion synthesis tool generates SVA's based on the simulation environment. These SVA's always hold true in the simulation environment. The counterexamples generated by formal can assist with constraints in the formal environment. If the SVA for which the CEX is found in the formal environment is true for the design, this indicates that the CEX found in formal should not have been found. This CEX found in formal should be investigated further. Root causing the CEX could lead to the following:

- finding an irrelevant constraint that should be removed
- fixing/changing an existing constraint
- adding a new constraint that should have been in formal but was not present

This technique of fixing up constraints in the formal environment can quickly help to ramp up the formal bench once a preliminary model is available.

D. Identifying Coverage Holes

Once counter examples have been generated in the formal environment, they need further investigation. If on visual inspection, we find that the SVA is not actually true for the design, this indicates that this was a coverage hole in the simulation environment for which we did not have stimulus. There are two paths that the SVA's may take at this point. If the implied coverage hole is not critical to the stimulus pattern, the verification engineer might choose to ignore this SVA and proceed. The other path of action would be to generate stimulus in the simulation environment. In this scenario, the property would be marked as coverage and we would try to verify that the property is in fact covered once the stimulus is created.

E. Golden Assertions and Reuse

The properties generated in the simulation environment and consequently proven in the formal environment have a high level of confidence. These SVA's are golden assertions and can be bound to the RTL module at other hierarchical levels. These SVA's are synthesizable and can be valuable when bound to other test benches. These SVA's can be valuable in identifying assumption mismatches between the various environments in which each RTL module is instantiated.

F. Bug Hunting

The SVA's generated from by the assertion synthesis tool enable the quick bring up and cleanup of the formal bench environment. This enables the formal environment to start bug hunting faster and more efficiently.

The formal environment identifies counterexamples of SVA's from the simulation environment. The counterexamples catch design bugs in two ways. One way would be a direct identification of the bug itself. The

other way would be a property, that we expect to hold true, is actually failing in formal; this indicates we have a bug in the design. The second case is more interesting because it indicates absence of stimulus in the simulation environment to catch the bug.

VII. RESULTS

The assertion synthesis tool was run on multiple units in our design. It was targeted to extract properties from control intensive modules in each unit where assertions are highly effective. The properties were generated by analyzing all passing simulation tests in the unit level regressions. These properties then are assumed to be assertions and ran in the formal model checking environment with Jasper [5].

Passing properties from formal under the correct set of constraints were treated as golden assertions and were integrated into the unit as well as the top level simulation regression tests. If any future tests cause them to fire, they are helpful to root-cause the failure and find additional bugs.

Failing properties are then debugged using a counterexample trace in the formal tool to determine the reason for the failure. The reason could be due to an incorrect constraint or a coverage hole missed due to simulation, or a design bug that was not found in simulation.

Table I shows the result of combining the flow in our first generation design where we have a mature simulation environment but a developing formal environment. Table II shows the result of combining the flow with our second generation design where we have an incremental update of the formal environment but a developing simulation environment (mostly topology changes and sequence updates). Both tables are based on formal run of 24 hours. The table II only shows results of one unit because our combining of the flow was interrupted due to simulator changes, as well as finding too many property failures.

TABLE I
RESULT ON FIRST GENERATION DESIGN

Formal Tool Result	Proven (passing)	Undetermined	CEX (failing)		
			Holes	Constraints	Bugs
Implication	golden assertions	No inputs from formal	Conflict due to constraint issue or coverage holes in stimulus		
UNIT1	193	237	13	13	0
UNIT2	205	531	17	16	0
UNIT3	165	375	20	15	0
UNIT4	38	44	3	5	0

TABLE II
RESULT OF SECOND GENERATION DESIGN

Formal Tool Result	Proven (passing)	Undetermined	CEX (failing)		
			Holes	Constraints	Bugs
UNIT2	151	357	301 (only debugged 37 of them, 174 are 1-2 cycle failures)		
			32	5	0

We can tell from above tables that, with a mature simulation environment, high quality properties are generated. It has less coverage holes and helped find more issues involving missing constraints in the formal environment. With a mature formal environment and a developing simulation environment, there were more coverage holes and quite a large percentage of noise (referring to these 1-2 cycle CEXs). Among the coverage holes, roughly 30% of them are important, and the rest (approximately 70%) are minor. Among the constraint issues, about 50% are due to modeling difference between the formal and simulation environments, and the remaining 50% are important. Further analyses are described in section below.

Our formal flow was mainly setup for bug hunting. Property convergence was 30-40%. However, undetermined properties could be further explored by choosing the right formal proof engines or bounded proofs to improve the convergence ratio.

A. Important Formal Constraint Issues Discovered

In the bring-up stage of formal, many of the formal constraints being developed were not matching the specification exactly. These incorrect constraints, if not cross-checked, would have hidden bugs (false positive), generated incorrect failures (false negative) and, in some cases, would have yielded more inconclusive properties. Cross-checking of constraints back into the simulation regression is recommended. But during the early constraint development stage, it will likely disrupt the routine regressions flow as the incorrect constraints will cause simulations failures. Debugging and correcting the constraints from simulation is an iterative, back-and-forth process where simulation regressions would need to be run multiple times until all of the constraints are passing.

Instead, when the assertion synthesis properties are derived from a relatively mature set of passing simulation regressions, they capture the design behavior in terms of properties from all the tests. These properties are run as assertions in formal and if the analysis of counter-examples revealed any property firing due to an incorrect constraint (not due to a simulation coverage hole), the process of debugging and correcting the constraint was self-contained in the formal environment and was relatively faster.

Some examples of constraint issues found are:

- A micro-architectural property that captured outstanding commands revealed a constraint issue where two outstanding commands of one type were sent using the same transaction ID. In the formal environment, it is tricky to constrain the command transaction IDs based on the tracking of pending and completed transactions. In our environment, there was a bug in managing of transaction IDs for one specific type of command that resulted in a property violation.
- Based on the coherency specification of our design, legal write snoop and read snoop types were limited to only certain values. The formal environment missed this important constraint.
- For some specific interfaces of our design, the command length should never cross 4 data beats. The formal environment missed this important constraint.
- For some read response transactions where no data is expected to be returned, the formal environment was not constraining it correctly and data was returned incorrectly. One micro-architectural property from the assertion synthesis tool revealed it.

B. Important Simulation Coverage Holes were Discovered at the Signoff Stage

Counter-example analysis also revealed properties firing under the correct set of constraints due to a coverage hole missed in simulation. Since the properties were derived from a set of mature simulation regression tests, they were a few, but very important ones. After careful analysis of these coverage holes, we marked them as coverage properties and modified our simulation verification environment to successfully cover them.

Some examples of the coverage holes found are:

- Consecutive strobes are allowed in latest AMBA ACE spec, but VIP limitations in simulation did not cover all possible cases. An assertion synthesis property that captures multi-bit signal behavior revealed that. We requested that the VIP team enhance the behavior to cover the case.
- When the coherent read request type is read shared, the destination must be a particular target, but the case was not covered due to incorrect random simulation constraints.

C. Unreachable coverage properties detected in formal, but that are covered in simulation, indicate limitations on formal.

Most of the properties were assumed to be assertions before they were run in formal. But some of the properties that were generated in unit level simulations, when violated at the top level, indicated important coverage holes missing from unit level simulation. We marked those relatively few properties as coverage properties and ran them in formal as a coverage target. We found most of them reachable/covered in formal with few exceptions. The ones that were unreachable revealed important limitations related to the configuration modes. Some of the configuration values are allowed by the specification to be changed dynamically when design is active (after reset), but the formal environment could not handle the dynamic change.

D. How Assertion Synthesis and Formal Could Benefit Each Other in Different Stages of Project

The benefits of assertion synthesis and formal are different when you are engaging them at different stages of the project. Here is the table that summarizes the benefits.

TABLE III
BENEFITS AT DIFFERENT STAGES

	Mature Project	Project in early stages
Quality of assertions generated	High In a mature project, the properties generated are based on a mature simulation environment. The mature simulation environment is likely to closely mimic design stimulus as dictated by the specification and hence cover the design more accurately. This leads to high quality assertions.	Low/Mediocre The properties generated at this stage may have several coverage holes and sorting through them to find the assertions could be tedious. The DV team also might want to just spend the effort in filling known stimulus holes and catch bugs via simulation instead.
Coverage contribution	Low/Mediocre Since the design is mostly verified, the coverage holes were probably already filled based on coverage analysis or bugs found. Thus coverage contribution at a mature project stage will be low, but any finds could be crucial.	High Since the design is at an early stage, the level_0 properties could pinpoint exact missing stimulus. The stimulus will help the DV team to drive up coverage.
New findings (stimulus holes/RTL issues)	Mediocre/High In a mature project new findings would be few. The simulation regressions are likely to have caught a lot of them. However, any issues found are of good quality because they were an escape from the simulation environment. The issues caught though likely to be few are of high value at this stage.	High This would be the stage when stimulus holes will be easy to catch. The creation of the stimulus itself could lead to RTL issues being caught in simulation. Assertion synthesis tool can also catch RTL issues through analysis of properties. However the DV team is likely to spend the effort to clean up all the functional fails found via simulation before analyzing assertion synthesis tool data which would indicate potential bugs that can be inferred from properties.
Formal contribution	High With the design mostly verified, the SVA's generated are high quality. These SVA's can be run through formal and help fix up the formal environment faster. This is because any counter examples found are likely issues with formal environment instead of a stimulus hole or RTL bug.	Low/Mediocre If the design/DV environment is not mature, there would be more coverage holes than assertions and hence the formal environment would not be able to benefit as much from the assertion flow.

VIII. LIMITATION AND FUTURE PLAN

Some of the limitations found and planning for improvement is as follows:

- Assertion synthesis is useful for identifying constraint issues in formal, but its benefit to augment or assist in writing those constraints automatically is imperfect due to the limited observability at the interface. With some additional capabilities in the tool and behavioral helper code at the interface, we are looking for ways to guide the tool and generate constraints automatically.
- Assertion synthesis cannot combine properties from different parameterized flavors of the IP. This results in the generation of properties for each flavor separately and some duplicate work when analyzing the violations in formal. Tools need to understand signal bit widths and behavioral differences due to parameters.
- Few properties need to be aware and qualified with data or command valid. Otherwise, they will violate on formal un-necessarily when valid is de-asserted.

ACKNOWLEDGMENT

We would like to thank Praveen Jain and Robert Utley for reviewing properties from the assertion synthesis tool and providing valuable feedback. We thank John Dickol and Andy Mcbride for supporting and encouraging us to finish this paper. We thank Mike Pedneau and Caio Campos for helping us on our formal verification effort. We thank Synopsys and Samsung for supporting us sharing the work at DVCON.

REFERENCES

[1] Synopsys BugScope, http://*www.atrenta.com*.
[2] Jonathan Bromley, "Double the Return from your Property Portfolio: Reuse of Verification Assets from Formal to Simulation", DVCON 2015.
[3] Shuqing Zhao, Shan Yan, "Automatic Generation of Formal Properties for Logic Related to Clock Gating", DVCON 2015
[4] Blaine Hsieh, Stewart Li, Mark Eslinger, Every Cloud – Post-Silicon Bug Spurs Formal Verification Adoption", DVCON 2015
[5] Cadence JasperGold, http://www.jasper-da.com/.