2017 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE

A Mutually-Exclusive Deployment of Formal and Simulation Techniques Using Proof-Core Analysis

Keerthikumara Devarajegowda, Jeroen Vliegen, Goran Petrovity*, Kawe Fotouhi*
Infineon Technologies AG
Email: *Firstname.Lastname*@infineon.com
*Cadence Design Systems
Email: goran@cadence.com, fotouhi@cadence.com

***Abstract*—In today's verification landscape, formal verification is primarily used as a bug-hunting engine in addition to simulation. Setups in which simulation and formal verification are applied in a mutually-exclusive manner, on the same design are rare. The difficulty is to validate the verification process, and to be sure that the two parts (simulation and formal methods) provide the verification coverage required. A detailed and uniform completeness metrics analysis, spanning both formal and simulation, is rarely applied on real designs. We propose an approach to adopt formal and simulation in a mutually-exclusive manner. First, we identified a set of formal-methods-friendly features and which will be formally verified. Next, we used JasperGold's ProofCore technology to extract structural coverage metrics from the formal proof and merged them with the RTL simulation code-coverage to produce a combined coverage-database. This allowed us to assess and confirm the scope of formal verification compared to simulation. The approach enabled us to reduce simulation effort as well as increase of the overall verification quality due to the combination of simulation techniques and formal methods.**

I. Introduction

To cope up with the rising complexity of "system-on-chips" (SOCs), various measures are being adopted by the semiconductor industry. These approaches include re-using existing components, adopting well defined and proven development methodologies. In-spite of these adaptations, exponential rise in design complexity threatens to nullify the effects of improved methodologies. Hence, chip designers and manufacturers are always looking for opportunities to effectively improve the overall design process.

A. Challenges

The biggest challenge of rising design complexity is to ensure functional correctness of the designs. According to a series of surveys conducted over the last decade in [1], functional verification requires on an average $50\,\%$ - $70\,\%$ of the overall development time. Hence, design and verification engineers are always looking for ideas to efficiently reduce the time required for verification, without compromising the quality of verification. UVM (Universal Verification Methodology), a simulation based technique has been established as a de-facto verification methodology. UVM allows verification engineers to build effective testbenches and addresses various verification issues such as simulator-independence, re-usability, scalability, etc. [2]. Simulation based verification techniques scale better with the growing design sizes but are not intended for exhaustive verification. Hence, design functionalities that need exhaustive coverage are hard to verify with simulation (ex: 21-bit Hamming Decoder). Application of simulation on such blocks might lead to possible bug escapes (corner-case/deep space bugs). On the other hand, Formal Verification (FV) offers exhaustive coverage and delivers high verification quality [3]. Formal-methods have emerged as a powerful technique, but still suffer from the limitations of mathematics. This is because, FV scales exponentially with linear growth of design size and require GBs of storage space and minutes/hours for property proof. Under these circumstances, simulation remains the primary choice for verification in the industry and FV is used on top of simulation as a bug hunting engine to investigate corner-case scenarios.

B. Opportunities

As mentioned earlier, both FV and simulation have their own set of strengths and weaknesses. Combining their positives into a unified verification flow offers several opportunities to tackle functional verification. In cases where FV is used on top of simulation, one can observe an overlap of verification efforts. Since FV offers exhaustive coverage, simulation need not be applied on parts that are already verified using FV. A verification approach that spans across both simulation and formal methods in a mutually-exclusive/complementing manner would offer various advantages. The term 'mutually-exclusive' highlights our approach in which different blocks of the design are either verified with FV or simulation such that their union provides complete verification of the DUT. A set of guidelines to select formal-friendly[1] blocks and coverage analysis spanning both the techniques are needed to realize the flow. With this work, we propose an approach to find an optimal mixture of FV and simulation on the same design.

The rest of the paper is organized as follows: Section II briefly talks about related work with respect to formal and simulation on the same design; Section III describes our proposed approach and various aspects of the flow; Section IV, V and VI elaborate the application of proposed approach on a real project; Section VIII provides the conclusion for our approach.

II. Related Work

FV and simulation are major verification techniques that are currently in use. A selection of the related works, in which the authors have worked in the direction of applying both techniques on the same DUT are examined below.

In [4], Yuan Lu et al. introduced a semi-formal verification methodology by tightly coupling both techniques. First, they selected buggy blocks of the design and verified them with formal methods. Valid traces from FV were used in simulation-based environment to verify system-level specifications. In addition, the traces generated by the formal engine were used to generate new traces for coverage analysis. In this closed-loop verification approach, both FV and simulation are applied on the same parts of the DUT. We suspect that the overall time required for the verification has been increased with this approach.

Rolf Drechsler and Görschwin Fey in their work titled "Improving Simulation-Based Verification by means of Formal Methods"[5] used FV to fill the gaps left (untested) by a simulative testbench. They first verified the DUT with simulation and used FV as a secondary verification technique. An interesting feature of this work is that the properties for FV were generated in an automated manner from the simulation traces. The generated properties were verified against the DUT in a formal tool. They concluded that the failing properties represent the gaps/holes in the simulation testbench. The approach is an ideal example for FV on top of simulation where FV is used to cover corner-cases.

Aritra Hazra et al. in [6] experimented with coverage management for simulation and formal property validation. They used a test plan language for unifying coverage goals for both techniques. According to them, low structural coverage numbers indicate low functional coverage, since all parts of the DUT are expected to contribute to the functionality. They evaluated the properties in both formal verification and simulation. The outcome of their work is as follows: With only formal methods, coverage achieved was $79\,\%$; With only Simulation, coverage achieved was $64.2\,\%$; and combined coverage was $93.8\,\%$.

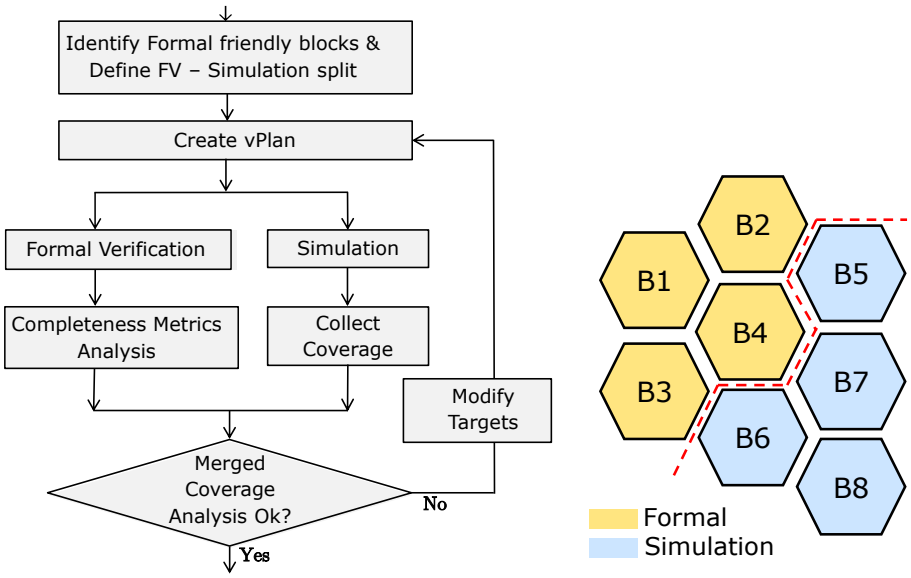
In [7] and [8], Michael Rohlender et al. work in the direction of combining results from different verification techniques (formal and simulation). Both works propose flows to combine results with respect to functional coverage only. In [8], the address range is divided hierarchically and verified based on the applicability of formal methods to them. In [7], the authors perform enhancements to metric driven verification by combining results from formal and simulation. They elaborate the importance of such a combination by considering the functional safety requirements of ISO26262. They point out the dangers of combining results from different verification environments.

The related works on the topic identify the need for a verification flow that spans both formal and simulation in an effective manner. In this work we use FV as a complementary technique to simulation. Additionally, our approach finds an optimum mixture of FV and simulation in a mutually-exclusive manner.

III. Proposed Verification Flow

The flowchart in Fig.1a depicts our proposed verification flow. The first step is to analyze the RTL implementation and identify blocks, sub-blocks or clusters of logic that are suitable for formal methods.

[1] Blocks/Sub-blocks/Clusters of the design that are more suitable to verify with formal verification are referred to as formal-friendly blocks.



(a) Verification Flow (b) Formal - Simulation split

Figure 1: Proposed approach for verification flow.
Our approach aims at finding an optimal mixture of formal and simulation in mutually-exclusive manner.

The RTL implementation (DUT) can be abstracted as an interaction between blocks (B1 - B8), with each block delivering a specified functionality as shown in Fig.1b. These blocks/sub-blocks are selected for the application of FV or simulation based on their suitability for formal-methods. A bird's eye view for selecting formal-friendly blocks are briefly tabulated in Table.I. More literature for selecting the formal-friendly parts of the circuit can be found in [9], [3]. The table must be considered only for reference. Several approaches have been proposed to formally verify FPU and sequentially deep designs by various authors [10], [11], [12]. In our work, we are trying to address the combination of both verification techniques on the same design. Application of FV on the identified blocks must return higher Return On Investment (ROI) when compared with simulation. The capabilities of modern formal tools must be considered while making the decision[2]. After categorising the blocks, an imaginary line separating the blocks based on their applicability for formal or simulation must be visible as shown in Fig.1b (dotted red line). The real-life RTL implementations may have functionalities that traverse several blocks. In such cases, logic clusters that are part of the functionality can still be considered and evaluated formally.

The next step is to create a unified Verification Plan (vPlan), considering both formal-friendly and simulation-friendly parts of the design. Creating a verification plan for FV has slight differences compared to simulation. In simulation, checkers and stimulus are tightly coupled since the checkers are validated only when required traces are stimulated by the input stimuli. Whereas in FV, checkers/asserts are defined to check generic behaviours that are independent of stimuli. However, assumptions (assume properties) are required on block-level input signals to provide legal input space for the property proof. Therefore, assumptions[3] drawn on incoming interface signals (from other blocks) must be asserted in simulation. The vPlan must be extended to include such assumptions. The vPlan is crucial for validating the functional completeness and hence, care must be taken to capture all required behaviours of the design. Also, proper metric must be set by the vPlanner for every checker/assertion as formal or simulation. This enables the visualization of overall and individual efforts required with respect to both FV and simulation.

After finalising the vPlan with checkers/assertions and cover-points, both UVM-based simulation and FV are applied on the respective blocks, independent of each other. At the system-level, test cases in simulation must be built to cover the required high-level behaviour of the design. A review and assessment of assertions is crucial for

[2]Abstraction techniques such as Counter Abstraction, Cut-points, Symmetric data, Tagging, etc. enable formal tools to handle much larger design blocks and also result in shorter run-times.
[3]Assume-guarantee: Assumptions made on interface signals, which are output signals of blocks verified with simulation.

confirming the functional completeness.

Design blocks/sub-blocks ideal for FV	Design blocks/sub-blocks not ideal for FV
Concurrent blocks, Control blocks, Data transport blocks	
Data transform blocks with less intense arithmetic operations (ex: Hamming encoder)	Data transform blocks with intense arithmetic operations
Sequential blocks with low sequential depth	Sequential blocks with very high sequential depth
General Examples: Bus interface units, Arbiters, Bus-bridges, DMA controllers, Interrupt controllers, Memory controllers, Token generators, Power management units, Proprietary interfaces.	General Examples: Floating point arithmetic units, Graphics shading units, DSP units, MPEG decoder.

Table I: Selection of formal-friendly blocks [9], [3].
Formal verification can handle complex to very large designs with several optimization techniques. The table is included to provide a general idea of formal-friendly blocks. [3].

Once the results from regression runs are available, coverage reports from both the formal tool and simulator are collected and mapped back to the vPlan. Structural coverage has been widely accepted as one of the metric for verification closure. Simulation based techniques have well defined coverage models that are uniformly adopted by various EDA vendors into their respective simulators. FV tool vendors are yet to reach such an agreement on coverage models. Hence, the coverage metrics used for formal methods depend on the tool being deployed. However, JasperGold provides coverage metrics that are in sync with that of simulation and there-by offer an opportunity to merge results into a single coverage database (db). The analysis of coverage results obtained from regression tests is crucial in order to confirm the split that is defined while creating the vPlan. Cover-db from the formal tool is merged with the cover-db from simulation to produce a unified coverage report. Modifications to the vPlan are performed depending on various factors to achieve the required coverage goals. When the coverage expectations are met, further steps in the design flow are performed.

IV. Application on a Real Project

The test vehicle (DUT) that was used for the application of proposed flow is a "Multi-voltage safety system supply" for Advanced Driver Assistance System (ADAS). The block diagram of the DUT, which is a mixed-signal design, is as shown in Fig.2. It is a SOC that receives input power supply from the car battery and provides a highly efficient multi-rail power supply, which is optimized for the use in ADAS. The system is expected to provide three different regulated[4] voltage outputs with programmable capability for voltage and current limitations. A Switched Mode Power Supply (SMPS) is used to maintain high efficiency for output rails. Internal supplies for the modules are provided by a central unit. The system consists of 3 regulators each operating at different voltage level as shown in Fig. 2. A monitoring block is used to handle under-voltage and over-voltage conditions. The Digital-Top-Level block represents the top module of digital logic including input/output pads. The digital core logic handles the product's state transitions and activation/deactivation of analog modules. Other features such as Serial Peripheral Interface (SPI) communication, watchdog and error monitoring are also handled by the digital module.

After analyzing the RTL implementation, the following blocks/sub-blocks and clusters of logic were selected as suitable for applying formal methods: SPI logic, Register files, SECDED unit (Single Error Correction Double Error Detection), Protection logic block and DEVCTRL (Device control) logic. Next, we created a unified vPlan capturing all the functionalities to be verified. Every checker/assert is assigned a proper metric as formal or simulation (If a checker is implemented with FV then the metric is formal). For FV we used the formal tool JaseprGold from Cadence Design Systems. JasperGold contains several automated 'Formal Apps', a Formal App can be defined as a pre-packaged solution addressing a specific verification problem. These formal apps read meta-data information in the form of IP-XACT or CSV and generate properties in an automated manner [13]. Formal apps FPV App, CSR App and COV App were used for our work on the topic.

[4]Voltage regulators are devices used to automatically maintain a stable voltage levels by using feedback loops.

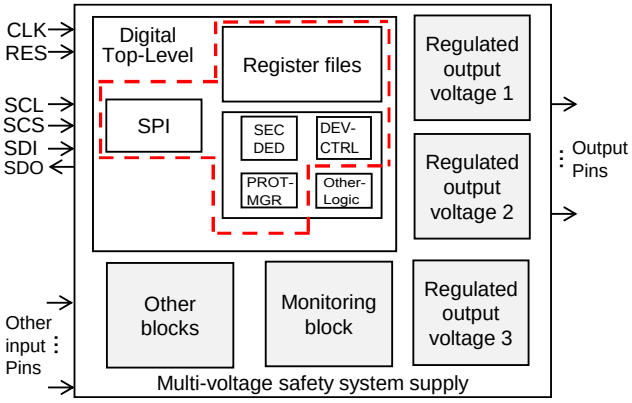


Figure 2: Block Diagram of Test Vehicle - DUT
Blocks enclosed by the dotted red line are identified as formal-friendly blocks.

A. *User-defined Property Verification*

We wrote manual properties to verify the SPI logic, SECDED logic, Protection logic and Device control logic. As already mentioned, the DUT is a mixed-signal design and reset for the digital module is released after a power-up sequence. During power-up, the output rails of the regulators are power sequenced to reduce the inrush of electrical current. In order to specify the reset of the digital module, it was required to bring the design to a known stable state. JasperGold allows to specify the reset signal from a supplied waveform [13]. Using dynamic simulation, we brought the design to a known stable state and loaded the waveform as a reset sequence into the formal tool. In addition to asserts/checkers and assumptions that were captured in the vPlan, we implemented cover properties to confirm the absence of over-constraining proof environment.

B. *Register files Verification*

"Multi-Voltage Safety System Supply" DUT consisted 62 registers, split between 4 different register files: R0, R1, R2 and R2_CFG. Register files consist of an array of registers with uniform structure and hence, properties could be generated in an automated manner. We used the CSR App for register file verification in which all properties were generated by the tool. The register file description was available in a meta-data format which we converted into IP-XACT format before applying the CSR App. Assert and cover properties, generated by the CSR App for different access policy modes were extensive and comprehensive. Interesting scenarios such as simultaneous bus read/write and direct (volatile) read/write are also covered and verified.

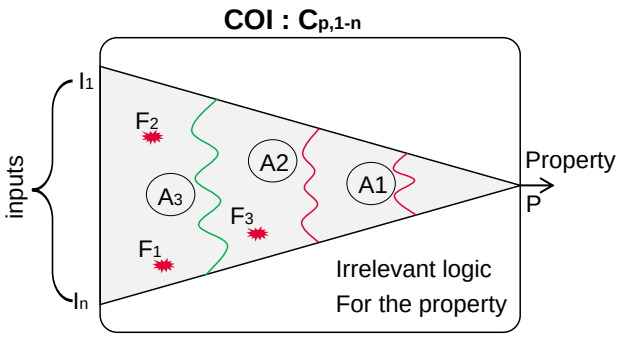
C. *Coverage collection*

In a properly constrained (absence of over-constraining) formal verification environment, completeness is implicit when the checkers are implemented to capture all required functional behaviours of the design. Nevertheless, we still need to determine if the property suite has captured all the required functional behaviors of the design [3]. We first review and assess the implemented checkers to confirm the functional completeness. Then, we collect structural information from the tool to confirm the absence of over-constraining, perform RTL code coverage and to identify coverage holes.

For structural coverage, we used the COV App and various structural coverage metrics offered by the tool were considered. JasperGold (v2016.03) supports block and statement coverage models (expression and toggle coverage models are forecast for the coming versions). After analyzing the available coverage models, following structural coverage metrics from JasperGold were considered:

- Dead-code and Stimuli coverage: Provide sanity check information (in addition to user-defined cover properties) to determine the presence/absence of over-constraining.
- ProofCore and Bounded coverage: Our approach uses ProofCore data as the main coverage metric for structural coverage. ProofCore analysis is elaborated in the next section. Bounded coverage shall be considered for properties with bounded proof.

V. PROOFCORE



Pseudo-Algorithm:

```
1. abstract the COI Cp,1-n : Cp,Ax ;
2. proof: prove (P) on the abstracted COI Cp,Ax;
3.    if (CEX) {
4.          Prove (P) on full COI Cp,1-n
5.          If( CEX ) {
6.             true negative;
7.             break; }
8.          else {
9.             x++;
10.            extend_the_abstraction;
11.            goto proof; }
12.   else if (P holds)
13.         ProofCore = PAm
```

Figure 3: ProofCore Analysis

ProofCore contains only that portion of the COI, which is in entirety required to prove or disprove the property. ProofCore is a structural coverage model in JasperGold.

ProofCore technology provides a comprehensive structural coverage model for discovering coverage holes in the property set. To understand the ProofCore technique, let us consider a property P as shown in Fig.3. $C_{P,1-n}$ represents the Cone Of Influence (COI) of the property P, $I_1 - I_n$ are on the boundary of the COI. A pseudo-algorithm for ProofCore analysis is listed in Fig.3. The validity (pass/fail) of property P is determined by the formal engines in the COI $C_{P,1-n}$. But, it is not effective to prove a property by considering all the signals on $I_1 - I_n$ boundary line. This is the brute-force approach (SAT without any optimization) and may lead to possible state-space explosion, depending on the number of state variables. Formal tools employ various state-space reduction techniques, in order to minimize the evaluation time required for the property proof. Such optimization techniques are internal to the tool.

JasperGold employs ProofCore technique, which is a state-space optimization that results in an effective COI for a property. Instead of validating the property P at input boundary line ($I_1 I_n$), JasperGold abstracts a line (A_1, need not be a straight line) that is closer to the point P and evaluates the property. If property P holds for the abstracted line, it holds for the entire COI $C_{P,1-n}$. The prior statement is in congruence with the proven theory, which is stated as follows: "if a property holds for an abstracted model, it also holds for the original model" [3]. In case of a CEX, the tool must check the validity of abstraction by checking if the same CEX can be found using the actual COI ($C_{P,1-n}$). If yes, the CEX is a true negative and the tool reports an error. Otherwise, the abstracted line A_1 is too coarse. JasperGold picks a line A_2 away from the first abstracted line and evaluates the property. If property P holds at this line, it holds for the entire COI ($C_{P,1-n}$). Otherwise, the abstracting line is again too coarse.

Let us assume that the property fails at abstracted lines A_1, A_2 and holds at the line A_3. The cone represented by $C_{P,A3}$ is the 'ProofCore' for property P. A section of the original COI between abstracted line A_3 and the input boundary $I_1 - I_n$, has no influence on the outcome of property proof. Also, fault injection (F_1, F_2) in the region between A_3 and $I_1 I_n$ has no influence on the property P. This part of the RTL logic is not included in proofCore and hence, should be covered by adding another property or checked for redundancy. However, inserting a fault F_3 inside the ProofCore results in a CEX for the property P. This confirms that the logic has been covered by the property P. [13]

ProofCore data of a property is the subset of original COI and contains only that section of the logic that is required for the pass/fail of a property. With ProofCore data, design mutation or error injection techniques become redundant. ProofCore data is a pessimistic approximation of a COI and thus provides higher confidence levels. ProofCore data helped us to identify coverage holes of our property suite. After carefully analyzing the ProofCore results, we wrote more properties to include the uncovered RTL logic.

To summarize the FV part, various blocks in the interface logic and register files are verified using the FPV App and CSR App respectively. Functional properties, sanity-checks and connectivity checks were implemented. The COV App was used to extract the ProofCore results. We loaded the coverage results from JasperGold to the Incisive vManager.

VI. SIMULATION

With the proposed approach, FV and simulation can be implemented in parallel and in a mutually-exclusive manner. Blocks, sub-blocks or clusters of logic that are verified with FV are no longer verified in simulation. However, since the register files are verified with formal methods, certain changes are required in the UVM-SystemVerilog (UVM-SV) simulation environment. This is because, various blocks/sub-blocks interact with register files through volatile as well as non-volatile register fields. When such blocks are verified in simulation, it is necessary to make sure that register fields are properly updated by the core logic (blocks/sub-blocks).

The UVM Register Abstraction Layer (RAL) enables high-level, object oriented models of register files. In a typical UVM-SV environment, RAL is used as an active model for register file validation and as a single-point reference for all SV models probing register field values. Register file validation is no longer required in simulation since they are already verified with the CSR App. In order to check core outputs, RAL needs to be in sync with the DUT registers. For this, we implemented a back-door mechanism using add_hdl_path() function calls and updated the register fields in RAL by peeking (peek() function call) into RTL registers upon a bus write/read transaction. After converting the RAL into a passive model, register-adapters and register-predictors are removed from the testbench. Next, we implemented checkers to check core outputs that modify the register fields. The checking mechanism is bidirectional and evaluated over a tolerance window of specified number of clock cycles. This is because SV reference models might predict the value of a volatile field earlier or later than the corresponding change in RTL registers depending on the testbench construction. After the changes, coverage results were collected from the Incisive simulator (NCSIM) and imported them into vManager. Overall, simulation metrics are reduced with respect to:

- Generation: Test cases or sequence items for blocks that are verified with formal are removed.
- Modeling: For blocks verified with formal methods, constructing reference models in UVM-SV is no longer required.
- Checking: It is imperative to remove checkers from simulation environment for the functionalities verified with formal methods.
- Coverage: Cover groups for all the blocks verified with formal methods including register files are removed from simulation environment.

VII. VISUALIZING COMBINED EFFORTS

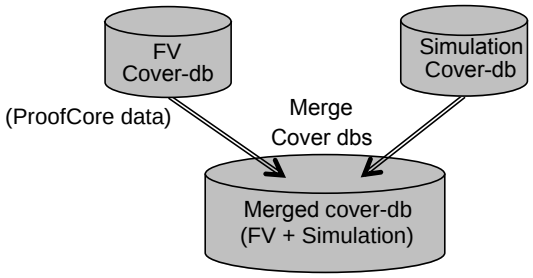


Figure 4: Merging coverage databases of Formal and Simulation.

In general, merging coverage from different environments (FV/simulation) can be challenging as there is no guarantee that the checker model is consistent in each environment. This makes it possible for one environment to cover an item and potentially trigger a bug, but if the appropriate checker is not present, it goes undetected. An appropriate checker may be present in another environment but the item does not get covered there, so the bug is not triggered. By merging the coverage from both environments the risk is coverage goals appear to be met and the project is signed-off with undetected bugs.

The problem is vastly reduced by taking a mutually-exclusive approach and partitioning the design into distinct functional blocks. It now becomes possible to clearly define the checker requirements for each functional block and then ensure they are fully implemented by the corresponding environment. A verification plan is key to achieving this where both checks and coverage from all environments can be viewed simultaneously to determine overall completeness, enforcing links between the specification and final implementation.

Real-life designs often contain functionalities that span several blocks. Following our approach, the functionality is either verified with FV or simulation depending on the decision made earlier (formal-friendly block selection). Complete isolation of blocks is not possible in such scenarios. However, this could potentially be addressed during RTL implementation by defining design guidelines ("Design for Formal"). Regardless, since the feature is only verified with one technique (FV/simulation) we still achieve mutual exclusiveness.

For functional completeness[5], pass/fail results of the property suite were mapped back to the vPlan. For structural coverage, ProofCore results from COV App were imported to the Incisive vManager. In simulation, we followed the existing standards by mapping the checkers pass/fail results back to the vPlan. Next, we collected coverage reports from NCSIM and imported them to the Incisive vManager. Then, coverage databases from both formal tool and simulator are merged into a single cover-db as depicted in the Fig. 4. After merging coverage results from both JasperGold and NCSIM, the overall coverage results were analyzed in vManager.

VIII. Conclusion

In this work, an approach that makes efficient use of both simulation and formal verification is proposed. The core idea of this approach is to use Formal Verification (FV) as a mutually-exclusive/complimentary technique to simulation. Design blocks are categorized based on the applicability of formal methods on them and selected formal-friendly blocks are only verified with FV. Such mutually-exclusive mechanism became possible, as formal tools provide coverage models similar to the ones established for simulation. Efforts needed in simulation are practically reduced. Reductions are realized in terms of generation, modeling, checking and coverage. With the use of automatic formal apps in which property generation is automated by the tool, efforts required in deploying FV is considerably reduced. This, in turn, increases the productivity of overall verification. By applying FV on more blocks, high verification quality can be reached. Since this work was pursued as a master thesis[6], we considered only the interface logic for selecting formal-friendly blocks. Since our simulation was small, reductions were also small, and potentially non-significant. We have not tabulated these results. Further work on larger designs would need to be undertaken to show the size of reductions.

Acknowledgements

We would like to thank Sebastian Simon and Alexander W. Rath for their valuable feedback and support during our work on the topic. Also, we would like to thank Carlo Del Giglio Matarazzo and Cadence Design Systems for the support with JasperGold. Finally, we would like to thank Mark Burton for providing valuable feedback and helping us to improve the quality of this paper.

References

[1] H. Foster, "Functional Verification Study," February 2017, [Online: accessed 20-February-2017].
[2] *Universal Verification Methodology (UVM) 1.2 User's Guide*, Accellera Systems Initiative, October 2015.
[3] E. Seligman, T. Schubert, and M. V. A. K. kumar, *Formal Verification, An Essential Toolkit For Modern VLSI Design*. Morgan Kaufmann Publishers, 2015.
[4] Y. Lu and W. Li, "A semi-formal verification methodology," in *ASIC, 2001. Proceedings. 4th International Conference on*, 2001, pp. 33–37.
[5] G. Fey and R. Drechsler, "Improving simulation-based verification by means of formal methods," in *Design Automation Conference, 2004. Proceedings of the ASP-DAC 2004. Asia and South Pacific*, Jan 2004, pp. 640–643.
[6] A. Hazra, A. Banerjee, S. Mitra, P. Dasgupta, P. P. Chakrabarti, and C. R. Mohan, "Cohesive coverage management for simulation and formal property verification," in *2008 IEEE Computer Society Annual Symposium on VLSI*, April 2008, pp. 251–256.
[7] S. R. Michael Rohleder, Clemens Roettgermann, "Enhancements of metric driven verification for the iso26262," in *DVCON Europe*, 2016.
[8] M. R. Clemens Roettgermann, Peter Limmer, "Achieve complete soc memory map verification through efficient combination of formal and simulation techniques," in *DVCON Europe*, 2015.
[9] H. Foster, L. Loh, B. Rabii, and V. Singhal, "Guidelines for creating a formal verification testplan," ser. DAC '06, New York, NY, USA, 2006.
[10] R. Kaivola and N. Narasimhan, "Formal verification of the pentium®4 floating-point multiplier," in *Proceedings of the Conference on Design, Automation and Test in Europe*, ser. DATE '02, 2002, pp. 20–. [Online]. Available: http://dl.acm.org/citation.cfm?id=882452.874523
[11] J. Harrison, "Floating-point verification." [Online]. Available: http://www.cl.cam.ac.uk/~jrh13/papers/iday.pdf
[12] A. Darbari and I. Singleton, "Industrial strength formal using abstractions," *CoRR*. [Online]. Available: http://arxiv.org/abs/1606.02347
[13] Cadence Design Systems, Inc., "Jaspergold apps user guide."

[5]In case of bounded-proof properties, one must assess if the feature needs to be verified in simulation.
[6]Master thesis is an academic work to be performed with strict deadlines in support of candidature for an academic degree.