2021 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE
OCTOBER 26-27, 2021

Democratizing Formal Verification

Shift-left by Bridging the Semantic Gap

Tobias Ludwig, LUBIS EDA GmbH, Kaiserslautern, Germany (*tobias.ludwig@lubis-eda.com*)
Michael Schwarz, LUBIS EDA GmbH, Kaiserslautern, Germany (*michael.schwarz@lubis-eda.com*)
Paulius Morkunas, TU Kaiserslautern, Kaiserslautern, Germany (*morkunas@eit.uni-kl.de*)
Silvio Santana, TU Kaiserslautern, Germany (*santana28@gmail.com*)
Dominik Stoffel, TU Kaiserslautern, Kaiserslautern, Germany (*stoffel@eit.uni-kl.de*)
Wolfgang Kunz, TU Kaiserslautern, Kaiserslautern, Germany (*kunz@eit.uni-kl.de)*

***Abstract*—In this paper we introduce *VERITAS*, a novel tool-flow enabling a "shift-left" of RTL verification and an automatic generation of formal *Verification IP* (VIP). The presented approach "democratizes" formal verification to a wider audience, resulting in shorter feedback cycles, reduced time spent on RTL verification and higher-quality designs.**

I. Introduction

The complexity of hardware designs is ever on the rise and with it the burden of design verification. Over the years there have been significant improvements in design and verification methodologies, however, the improvements have been merely sufficient to offset the increase in design complexity.

In 2020, 68% of ASIC projects finished behind schedule, with only 32% achieving first-silicon success, i.e., over two-thirds required at least one respin. Nearly 50% of the respins were caused by logical or functional bugs, a figure that has remained virtually unchanged since the early 2000s [1].

An important measure to reduce time to market is not to wait until a physical hardware (HW) prototype of the computing platform is available but to begin developing system software (SW) much earlier, i.e., to "shift-left" SW development on the timeline. *Virtual Prototypes* (VPs) are an effective shift-left approach to remedy some of these issues, enabling early design elaboration and bug discovery. Unfortunately, the semantic gap between the *Electronic System Level* (ESL) and the *Register Transfer Level* (RTL) prevents a direct transfer of verification results between the two model domains, which leads to duplication of work and additional test and debug efforts.

Hope to bridge this gap lies in formal approaches [2][3][4][5]. They are becoming more and more important to keep up with rising complexity demands. The expected benefits of these approaches are - to name only a few - *better design quality*, *improved system safety*, and *enhanced cybersecurity*. As shown in [6], using formal verification early in the design process results in a 25% shorter time to market. Currently, a wider adoption of formal methods in the industry is primarily limited by educational issues, e.g., a lack of training of engineers and a general hesitation with engineers to engage in formal techniques when they expect the learning curve to be steep [7].

The paper is structured as follows: In Sec. II we provide an overview of the current VERITAS flow and provide insight on the technical and theoretical backgrounds. In Sec. III we are going to relate our work to already existing approaches of the industry. Sec. V shows how we extended VERITAS to deal with pipelined designs and Sec. VI provides results on existing and the extended approaches.

II. Overview

We introduce *VERITAS* [8], a tool flow that guarantees formal soundness between ESL and RTL and, thus, enables a shift-left of general functional verification by moving HW verification to a higher abstraction level. In addition, by automatically generating a formal RTL *Verification IP* (VIP) from ESL descriptions, the entry hurdle to formal methods is reduced considerably, opening them to a wider audience, which effectively 'democratizes' them. Short feedback cycles reduce time spent on RTL verification and lead to higher-quality designs. Figure 1 shows the proposed verification approach, building upon existing flows like *Unified Verification Methodology* (UVM). Starting from a set of use cases (e.g., constraint-random stimuli or precomputed input sequences), a behavioral *Golden Model*, e.g., specified in SystemC-TLM, is checked for correctness with a simulation-based approach. Ideally, the Golden Model is modeled at the transaction level and is considered, in the remainder of this work, to be an ESL model.
In a traditional approach, a time-costly re-verification of the RTL model of the *Design Under Test* (DUT) against the same use cases would be required for building confidence that the Golden Model and the RTL are behaviorally equivalent. Our approach removes the need for a complete functional re-verification of the DUT, saving valuable RTL verification time. In addition, it allows to cover the entire RTL design behavior with a complete set of formal properties.

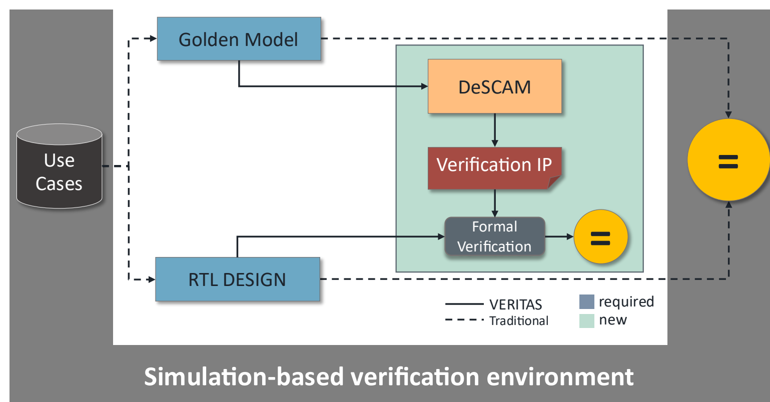

Figure 1 VERITAS flow vs. traditional flow

The VIP covers the design intent described by the ESL model as a *complete* set of formal properties (e.g., as SVA), wherein each property covers the hardware behavior in terms of a transactional operation between automatically inferred states, e.g., different stages of a pipeline. Instead of re-running the simulation-based verification with the DUT, the design is verified formally using the VIP. If all properties hold on the DUT, the design is correct-by-construction, w.r.t. the Golden Model.

The basic ingredients to the new *VERITAS* flow are:

- *VERITAS*, an EDA tool that automates VIP creation for a target design in a chosen verification language, thereby significantly reducing required verification expertise and effort.
- Standard industrial formal verification tools and verification techniques.
- Path-Predicate Abstraction (PPA), a mathematical model that allows to establish formal relationship between ESL and RTL based on formal properties. Therefore, verification results obtained at the ESL also apply at the RTL.

As a key contribution, we introduce a new type of VIP that is used to verify *pipelined designs*. It enables a microarchitectural design space exploration at the RT level. Designers can experiment with different pipeline latch/register configurations to optimize the circuit timing, while the VIP always ensures a functionally correct behavior that is proven by formal checks. A particular benefit of the VIP is that it allows to detect pipeline-related issues, e.g., bugs in forwarding units, erroneous stalling, or issues with pipeline optimizations. Such bugs are hard

to find and debug with traditional simulation-based approaches. Running the VIP produces a minimal trace that pinpoints the root cause of the bug. The VIP can be used with any implementation of static pipelining, not only in processors but also elsewhere, e.g., in communication infrastructures.

III. RELATED WORK

VERITAS builds upon 15 years of research. Due to the limited space, we only briefly cover the theoretical foundations by pointing out the main ideas and major publications. [2][3] introduces *Path-Predicate Abstraction* (PPA), a mathematical model to establish a formal relationship between RTL and ESL models, making the ESL a sound abstraction of the RTL. The term *soundness* relates to the fact that any verification results obtained for the PPA model also hold on the RTL. Hence, proving the absence of a bug on such a model also means that there is no bug at the RT level.

[4][5] presents a methodology to specify executable ESL models (e.g., with SystemC TLM), which adhere to the semantics of the PPA. Furthermore, an algorithm to transform such models into a VIP is provided. Proving the VIP on the RTL establishes a sound relationship and proves the functional equivalence w.r.t. the ESL. We briefly cover the core idea of this approach in Sec. IV. [14] uses the PPA model to perform aggressive RTL power optimizations and [13] describes a novel hardware generation framework based on PPA models. However, they do not aim to generate a verification IP.

The generation of the VIP for pipelined designs is based on a verification technique called *Symbolic State Quick Error Detection* (S^2QED) [9]. S^2QED proves that every instruction executes independently of the previous ongoing instructions in the pipeline, i.e., independently of its context. The computational model of S^2QED consists of two identical and independent instances of the processor under verification which are constrained to execute the same instruction, at an arbitrary time point.

In [15] the technique is extended to ensure a complete formal coverage along with the proof. In the scope of this work, we remove the manual efforts and the requirements on expert knowledge, by automating the generation of the S^2QED properties. S^2QED is mainly developed to verify processor designs. We extend the S^2QED idea for other types of designs.

In [6] the idea of generating properties from a formalized specification as part of a hardware generation framework is described. In contrast to our approach, the generation in [6] does not start from an executable model. Instead, the formalized specification is a structural description. The approach mainly targets processors or processor-like designs. VERITAS, on the other hand, has no restrictions on the target areas.

IV. VERITAS BACKGROUND

In this section, we describe the VERITAS design flow. The goal is to provide an intuition of how the executable behavioral model can be used to generate a VIP, and how the VIP is used to establish a sound formal relationship. After describing the transformation steps, we cover the core elements of the verification IP. What is not covered in this work are the domain-specific source code optimization techniques that ensure that the generated VIP is human-readable and concise, and that it fully describes the design intent.

A. Under the hood of VERITAS

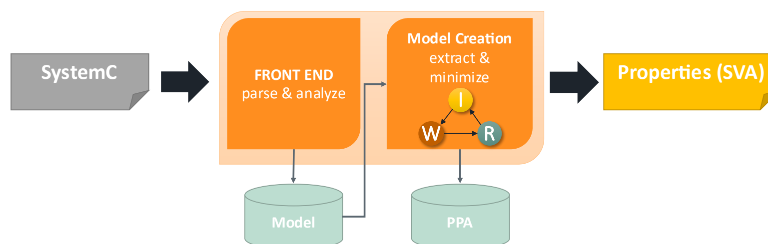

Figure 2 ESL to Properties

Figure 2 provides an overview of the flow, starting with an executable behavioral model. Ideally, the model is a transaction-level model that abstracts from low-level RTL details. Such a high-level model, e.g., in SystemC or

behavioral SystemVerilog, may already exist from concept engineering or architecture design, or it can be created based on the specification.

1. The behavioral model is parsed and analyzed by VERITAS. In this step, necessary behavioral and structural information is detected and stored in a data structure called *Model*. Additionally, the tool analyses if the provided model fulfills the requirements of a "designable" subset, i.e., it does not use features like dynamic memory allocation.
2. From the Model, a path-predicate abstraction (PPA) is extracted, consisting of "important" states and transitions between them. It has the form of an FSM capturing the design intent specified by the Model.
3. The PPA is minimized regarding the number of important states and transitions to create a concise and human-readable VIP based on the PPA.

B. *From source code to PPA*

```
1: SC_MODULE(encoder) {
2: // Constructor and declarations
3: void fsm() {
4:   while(true){
5:     bus_in→read(mode, data);
6:     if (mode == send) {
7:       enc(data);
8:       bus_out→write(data);
9:     }
10: }}};
```

Figure 3 Simple SystemC module

Figure 4 Example PPA

Figure 5 RTL implementation

Figure 3 show a simple example of a behavioral model (in SystemC), that encodes data received from an input port *bus_in* and, upon completion, transmits the encoded data via an output port *bus_out.* The main task of the tool is to extract the PPA as shown in Figure 4. If and only if the PPA is successfully extracted it can be used as a foundation for establishing a formal relationship between ESL and RTL, enabling a shift-left of verification and a full formal coverage of the functional behavior.
The PPA is composed of two important states read *R* and write *W*, four transitions and one reset operation. In this case, the important states directly result from the communication calls of line 5 and line 8. Different types of communication interfaces are supported; however, the provided example only depicts blocking calls. Such calls are modeled with wait operations that enforce retaining the current important state within PPA until external event occurs, i.e., triggered handshake signal. Upon reception/transmission of the data the execution continues. The behavior related to the transition labeled *encode* results from line 7.

C. *Operational Properties*

In this section, we introduce the basic building blocks of the VIP, consisting of operation properties and refinement functions/macros.

```
1: property encode(length);
2:  // Freeze variables
3:  int frz_data;
4:  // Freeze values
5:  t ##0 hold(frz_data, bus_in_data()) and
6:  // Triggers
7:  t ##0 state() == R and
8:  t ##0 bus_in_mode() == send
9: implies
10:  t_end(length) ##0 state() == W and
11:  t_end(length) ##0 bus_out() == enc(data_0)
12: endproperty;
```

Figure 6 Generated operation property

```
1: function int bus_in_data()
2:   return {
3:     $past(unit/data_in,3),
4:     $past(unit/data_in,2),
5:     $past(unit/data_in,1),
6:     unit/data_in};
7: endfunction;
```

Figure 7 Non-trivial refinement

```
1: function int bus_in_data()
2:     return unit/data_in;
3: endfunction;
```

Figure 8 Trivial refinement

Figure 6 shows a property named *encode* that is generated for the transition from *R* to *W* in Figure 4. The property is an *Interval Property Checking* (IPC) property and is expressed in standard SVA. An IPC property has the form of an implication between an *assumption* and a *commitment*. The assumption describes a certain start state and

trigger events. The formal engine verifies that for any state fulfilling the assumption the commitments hold on the model. In the remainder of the work, we call such a property *operation property* because it specifies an important operation of the design. Each transition between two states in the PPA results in an operation property.
Figure 6 verifies the transition from *R* to *W* and that the output *bus_out* is set to the correct value. Line 3 introduces the intermediate variable *frz_data* that is used in line 5 to store the received value for later use. The function *hold()* is generated by *VERITAS* and hides the underlying complexity of storing a value at a certain timepoint. Lines 7 and 8 form the assumption of the property. We assume that the design is in the important state *R* and the received mode is equal to *send*.
The engine proves in line 10 that the end state is *W* and in line 11 that the output *bus_out* is set to the correct value. The value depends on the received value at timepoint *t0* and a user specified function *enc()*. The encoding function is part of the golden model and VERITAS takes care of providing the according SVA function for *enc()*.
The function *t_end()* is another function that hides complexity for the user.

D. Refinement of the VIP

The length of the property is specified by providing a value for the parameter *length* in line 1. Figure 5 shows the state machine of a possible implementation. We observe that the transition from *R* to *W* takes two clock cycles. In this case, the parameter *length* is chosen to be 2. The function *t_end()* dynamically adjusts the proof depending on the length of the operation. The user does not edit the generated properties except for specifying the length.
However, many RTL implementation details are unknown and kept abstract at the ESL, and the user needs to bridge this abstraction by "refining" the macro-functions. Figure 7 and Figure 8 show two examples of different refinements for the macro-function *bus_in_data*(), returning a 32-bit integer. Figure 8 may be considered a trivial refinement because it only specifies a reference to one concrete signal of the RTL design (note, trivial refinements can be automated). More interesting, however, is the refinement in Figure 7, which enables a sequential production of the 32 bits. Such implementation details are decided by the designer and need to be reflected by a corresponding refinement of the macro-functions of the generated VIP.

The flow provided by VERITAS enables anyone to use formal verification and thereby benefit from all advantages formal verification has over classical simulation. Aside from refining the macro-functions, no technical expert knowledge is required. Another benefit of the generated VIP is that it ensures a full formal coverage of the design. The properties build a gapless chain – the ending state of one property coincides with the starting state of the next one. Additionally, the assumptions of the properties are structured in a way that it is ensured that all possible trigger conditions for operations are covered. The operation properties cover the entire reachable state space of the design. Hence, the VIP will uncover any functional bugs that exist within the design. Lastly, it is important to remember that the sound relationship of the ESL and RTL allows a shift-left of the verification towards higher abstraction levels.

V. EXTENDING VERITAS FOR PIPELINING

In practice, it is sometimes necessary to pipeline the execution of certain operations to meet performance requirements. As it is, the approach presented in Sec. 3 works well for sequential designs. However, verifying a pipelined design with these types of properties is troublesome. Figure 9 illustrates the problem. In a sequential design, only one operation executes at a time, whereas in a pipelined design two or more operations may be active simultaneously. The time points of the commitment of one operation "overlap" with those of another one (indicated by the red box). This results in complex refinements because the macro-functions must be refined to reflect every possible execution case.

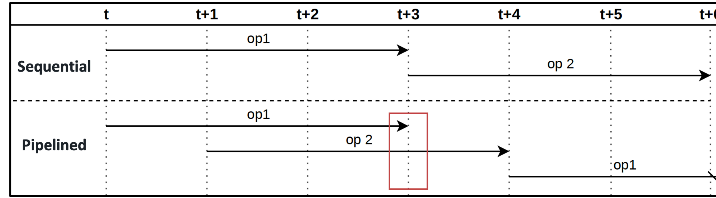

Figure 9 Sequential vs Pipelined Operations

To solve this issue, we extend VIP generation using the S^{2}QED technique [9] mentioned above, in the following way. We begin with the flow as presented in Sec. II, however, now we generate two sets of formal properties called base properties and relaxed properties, respectively:

- The base properties prove the execution of a single operation (e.g., an instruction) in the pipeline, starting from a flushed/empty state.
- The relaxed properties prove that the design has the same functional behavior independent of the pipeline state, i.e., in all pipeline contexts.

The base and relaxed properties are expressed in a special form of operation properties. They span over multiple important states instead of just describing a single transition between two important states. As part of the design flow, the user adds information about the desired pipeline stages directly into the ESL model (note that this does not change the sequential nature of this model). It is important that the ESL model remains sequential. Only then a shift-left of verification is attractive due to increased simulation performance. The algorithms automatically compute the required information for generating the properties.

A. *Base properties*

Figure 10 shows a pseudo-code example of a base property for a 4-stage processor. It extends the operation property by introducing multiple timepoints (one for each stage of the design) and an *reference_state()* macro-function (see line 8) which must be refined by the user to specify in which situations the pipeline is empty. The base property proves functionally correct execution of a single operation. For example, the property shown in Figure 10 proves the correct execution of a first add instruction as it moves through the pipeline. If there is a functional bug in the computation, e.g., an incorrect addition, this property fails and provides a minimal counterexample revealing the bug.

```
 1: property base_add();
 2:  // Freeze values
 3:  t_IF ##0 hold(frz_pc, pc) and
 4:  t_EX ##0 hold(frz_result, ex_result) and
 5:  t_EX ##0 hold(frz_dest, ex_dest) and
 6:  // Triggers
 7:  t_IF ##0 state == IF and
 8:  t_IF ##0 reference_state() and
 9:  t_ID ##0 instr == ADD
10: implies
11:  t_ID ##0 pc == frz_pc + 2 and
12:  t_EX ##0 ex_result == reg[rs1] + reg[rs2] and
13:  t_EX ##0 ex_dest == getDest(instr) and
14:  t_WB ##0 wb_result == frz_result and
15:  t_WB ##0 wb_dest == frz_dest and
16:  t_DONE ##0 update_regs(frz_result, frz_dest)
17: endproperty;
```

Figure 10 Example of a base property

```
 1: property relaxed_add();
 2:  // Freeze values of instance 1
 3:  t_WB_i1    ##0 hold(frz_regs_i1,     regs_i1) and
 4:  t_DONE_i1 ##0 hold(frz_regs_i1_new, regs_i1) and
 5:  // Constrain reference state on instance 1
 6:  t_IF ##0 reference_state() and
 7:  // Same important state and fetched instruction
 8:  t_IF ##0 state_i1 == IF &&  state_i2 == IF  and
 9:  t_IF ##0 instr_i1 == ADD && instr_i2 == ADD and
10:  // Consistent general-purpose registers before WB
11:  t_WB_i2 ##0 regs_i2 == frz_regs_i1 and
12: implies
13:  // Consistent general-purpose registers after WB
14:  t_DONE_i2 ##0 regs_i2 == frz_regs_i1_new
15: endproperty;
```

Figure 11 Example of relaxed property

In lines 3 to 5, values are captured at distinct time points. For example, the value of the program counter, *pc*, is captured in a variable *frz_pc*. The computation of the next PC in line 11 is based on the value of the PC at the *Instruction Fetch* (IF) stage. The same idea applies to results that are computed in line 12 and line 13. The property proves, in line 16, that at the end of the add operation, the registers are updated to the correct values.

B. *Relaxed properties*

Just proving the base property of an operation is not sufficient for functional correctness, because it does not cover the cases where several operations are in execution simultaneously. As explained in Sec. III, the core idea of S²QED is to include two instances of a design in the computational model – one is a constrained instance and the other one has a free initial state. The name "relaxed" relates to the fact that the second instance is not restricted in its initial state. This allows to prove the absence of pipeline-related bugs.
The relaxed property (see Figure 11) uses two sets of timepoints and macro functions for the instances (indicated by the suffixes *_i1* resp. *_i2*). Lines 8 and 9 in the property ensure that the two instances are in the same important state and fetch the same instruction. The consistency assumption in line 11 ensures that the architectural state of both designs is equivalent before the constrained instance writes back its results. This assumption is necessary to be able to prove in line 14, that both instances end up in the same architectural state after the write back.

There are two main causes of a failing relaxed property:

- Incorrect refinement of the *reference_state()*: Debugging the counterexample reveals that the constrained instance starts from an incorrect starting state (e.g., the *reference_state()* is not an invariant or it is over-constrained). The user needs to modify the macro-function and repeat the proof for the base and relaxed property. This works also as a sanity check for choosing the correct starting state for the base property.

- Functional bugs in the design: The property fails and produces a counterexample in which the execution of an operation depends on the pipeline context, e.g., because of unresolved structural or data hazards. The task of the designer is to understand why the two instances end up in different states and adjust the DUT accordingly.

If proving the relaxed and the base property is successful then this indicates that the corresponding operation is executed by the pipeline correctly in all contexts, with no functional bugs. Lastly, having this VIP as part of the design flow allows performing aggressive changes to the pipeline structure in a "try and error" manner. The properties can be verified fast enough so that different design variants with different operation timings may be explored without compromising functional correctness.

C. *Target areas and technical limitations*

The target area for using the generated VIPs are control-dominated designs like, e.g., a bus or a memory controller, and processor-like designs with static pipelines. Formal verification has its limitations in data/signal processing applications, and these limitations, naturally, apply to the presented approach as well. The next evolution of the VIP will address techniques to automatically black-box verification-expensive arithmetic computations, in order to make the presented technique applicable for data-centric designs as well.
VERITAS currently only accepts ESL model descriptions that adhere to the language subset as specified in [4] (comparable to SystemC TLM 1.0). Future improvements of the tool will aim at extending the subset to include the full SystemC synthesizable subset as well as accepting other formats (e.g., Behavioral SystemVerilog or SystemC TLM 2.0).

VI. RESULTS

All experimental results were obtained on an Intel Core i7 @ 3GHz with 32 GB of RAM. All property checks were conducted with the commercial property checker OneSpin 360 DV [10]. Table 1 shows the results for three case studies. *CV32E4P* is an industrial and open-source RISC-V processor [11] formerly known as R1SCY. *Amba High Speed Bus* (AHB) is an implementation of the open-source SoC interconnection architecture [12]. *SONET/SDH* is an industrial long-distance protocol implementation, that is not pipelined, but is still included here to show the benefits of the regular approach. The goal of the case studies is to evaluate the simulation speed-up w.r.t. to a set of predefined use cases. The results show that the generated VIP scales for real-world designs.

Table 1 Experimental results

Design	LoC		Use-case simulation		Verification IP	
	ESL	RTL	ESL	RTL	# of properties	Proof time
CV32E40P	856	6600	<0:01min	0:43min	22	156:00min
AHB	603	2630	0:07min	34:50min	17	68:00min
SONET/SDH	780	27000	<0:02min	9:01min	22	23:00min

The LoC column in Table 1 shows the respective lines of code for the SystemC-TLM models and the RTL implementations. We see a significant reduction of code size, resulting from the abstraction between RTL and ESL. In the conducted case studies, the obtained speed-up is in the range of 40x to 300x, based on the use cases with the smallest speed-up. The generated VIP allows to prove functional correctness of the RTL within a reasonable amount of time. Note that it is only necessary to perform the formal proof of the VIP once to be able to transfer the results of all use-case simulations. The proof time of the processor and the AHB bus are the combined numbers for base and relaxed properties. In practice, base properties finish in a fraction of time compared to the relaxed properties. For all three case studies, a bottom-up approach is chosen to show that VERITAS compatible (designable) SystemC models provide enough flexibility to be used as entry point Golden Models for real world RTL designs. The goal is to show that the designer is not limited in his decision making by the generated VIP.

VII. Conclusion and Future Work

In this paper we introduced *VERITAS*, a novel verification flow that democratizes formal verification, opening it up to a wider audience and paving a new way to approach today's verification complexity issues. The experimental results clearly show the benefits of the approach by enabling a "shift-left" of verification to higher abstraction levels. Future work will focus on generating the VIP for safety and security applications as well as extending the parser to other languages.

References

[1] Siemens Digital Industries Software, "The 2020 Wilson Research Group Functional Verification Study", 2020

[2] J. Urdahl, D. Stoffel, M. Wedler and W. Kunz, "System verification of concurrent RTL modules by compositional path predicate abstraction," DAC Design Automation Conference 2012, San Francisco, CA, USA, 2012, pp. 334-343, doi: 10.1145/2228360.2228422.

[3] J. Urdahl, D. Stoffel, J. Bormann, M. Wedler and W. Kunz, "Path predicate abstraction by complete interval property checking," Formal Methods in Computer Aided Design, Lugano, Switzerland, 2010, pp. 207-215.

[4] T. Ludwig, J. Urdahl, D. Stoffel and W. Kunz, "Properties First—Correct-By-Construction RTL Design in System-Level Design Flows," in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 39, no. 10, pp. 3093-3106, Oct. 2020, doi: 10.1109/TCAD.2019.2921319.

[5] T. Ludwig, M. Schwarz, J. Urdahl, L. Deutschmann, S. Hetalani, D. Stoffel and W. Kunz Property-Driven Development of a RISC-V CPU. In Proc. Design and Verification Conference United States (DVCON-US '19).

[6] K. Devarajegowda and W. Ecker, "On generation of properties from specification," 2017 IEEE International High Level Design Validation and Test Workshop (HLDVT), Santa Cruz, CA, USA, 2017, pp. 95-98, doi: 10.1109/HLDVT.2017.8167470.

[7] H. Garavel, M. H. ter Beek, and J. van de Pol. "The 2020 expert survey on formal methods." *International Conference on Formal Methods for Industrial Critical Systems*. Springer, Cham, 2020.

[8] LUBIS EDA GmbH, https://www.lubis-eda.com

[9] M. R. Fadiheh et al., "Symbolic quick error detection using symbolic initial state for pre-silicon verification," 2018 Design, Automation & Test in Europe Conference & Exhibition (DATE), Dresden, Germany, 2018, pp. 55-60, doi: 10.23919/DATE.2018.8341979.

[10] OneSpin Solutions, https://www.onespin.com

[11] Open HW Group, https://www.openhwgroup.org

[12] ARM, "Amba Highspeed Bus", https://developer.arm.com/architectures/system-architectures/amba

[13] L.Deutschmann, J.Schauss, T.Ludwig, D.Stoffel, W.Kunz: Operation-Level Synthesis, 24. Workshop Methoden und Beschreibungs-sprachen zur Modellierung und Verifikation von Schaltungen und Systemen (MBMV'21), Virtual Event, Germany, March 2021.

[14] S.Udupi, J. Urdahl, D. Stoffel, and W. Kunz, Exploiting hardware unobservability for low-power design and safety analysis in formal verification-driven design flows, IEEE Transactions on Very Large Scale Integration (VLSI) Systems, June 2019.

[15] K. Devarajegowda, M. R. Fadiheh, E. Singh, C. Barrett, S. Mitra, W. Ecker, D. Stoffel,and W. Kunz, Gap-free processor verification with s^{2}qed and property generation, in 2020 Design, Automation Test in Europe Conference Exhibition (DATE), Grenoble, France, March 2020