Hardware Verification In An Open Source Context

Ben Marshall

Department of Computer Science, University of Bristol, ben.marshall@bristol.ac.uk

***Abstract*—The last few decades have seen the complexity of commercial hardware designs increase by multiple orders of magnitude. This has driven corresponding increases in commercial tool capability and the development of industry standard methods to drive them. Over the same time period, open source hardware development has lagged severely behind in terms of the scale of attempted designs, as well as the tooling and methodology capability to realise them. This capability gap is particularly acute for verification flows. In this paper, we try to explain how this capability gap appeared, and what it means for a nascent open source EDA community. We also survey the state of the art in commercial verification techniques, and list alternative open source tools where available. Where open source alternatives are lacking, we make suggestions for closing the gap. We also discuss several human factors and challenges relevant to verification in an open source context, and suggest a change in mindset is needed to make open source hardware designs more trustworthy.**

***Index Terms*—Open source, hardware, verification, EDA**

I. Introduction

Complexity in hardware design has increased exponentially over the last few decades. Driven by Moore's law, the complexity of VLSI designs, and the number of distinct components on a chip has increased significantly. Taking advantage of Moore's law has required enormous investment in EDA tooling at all stages of a design process. Particularly, since the cost of re-fabricating a chip if a defect is found post manufacture has increased significantly as process sizes have shrunk.

With few exceptions, much of this development has been driven by commercial companies like Cadence, Synopsys and Siemens/Mentor, along with their acquisitions. The tools these companies offer are closed source and very expensive to licence. The size of designs being pushed through these tools has also motivated industry to develop standard methods for managing the complexity of functional verification.

In contrast, open source hardware (OSH) development has lagged significantly in terms of available tooling (with some admirable exceptions discussed later) and hence the size and complexity of designs attempted. This is a particular problem for verification tooling and methods. The relatively small number of OSH designs actually manufactured has meant less motivation to make the same investments in tooling and methods seen in industry.

Recently, this has begun to change. A series of high-profile feature, bug and security disclosures [1], [2], as well as long lived anxiety as to the trust-worthiness of commercial hardware designs, has motivated the building of auditable, open-source alternatives. As a result, Open Source (OS) tools for hardware synthesis, and even complete OS FPGA toolchains have started appearing. The next step will be to enable easy taping out of OS designs in high volume manufacture, something already being worked on.

However, all of these developments miss a crucial part of the hardware development picture: functional verification. The OSH community has not yet faced the challenges of verification at scale which commercial companies have. Further, while there is considerable emphasis on creating and implementing OS designs, comparatively little has been said about the state and necessity of their verification. Though there are some exceptions, in our experience we find verification is currently a second class endeavour next to design in the context of OSH.

In this paper, we give some background as to how this capability gap between OSH and software arose, and how the community can work on closing it. The rest of the paper is organised as follows: Section II-A provides some context on how OSH got "left behind" in terms of its tooling support. Section II-B includes a discussion of the current main approaches to hardware verification used in industry, and the motivations for their inception. Section II-C gives a short comparison of capabilities between commercial and OS alternatives. Section III then lists a set of open challenges in OSH verification. These are themed around tooling, methodology and human factors. Section IV concludes the points made in the paper.

II. Background

In this section, we give some context about the current capability gap in verification tooling and methodology seen between the OS community and industry.

A. The capability gap

Fundamentally, the need for more capable EDA tooling is driven by the need or ability to realise larger and more complex hardware designs. As manufacturing process nodes have shrunk, so the cost of making late stage design alterations has increased [3]. When the cost of spinning a chip dominates the cost of implementing a design, it becomes necessary to spend more effort on ensuring its correctness. This has put considerable back pressure on early stages in the development cycle to find all functionality bugs as early in the design stage as possible.

Given these driving forces behind tool development, it becomes clear why such a capability gap has arisen between industrial and OS communities. The OS community has not yet attempted the same complexity or scale of designs seen in industry. While there have always been OSH projects, only a tiny fraction have found their way into actual silicon. Even

This work was presented at the first Workshop on Open-Source Design Automation (OSDA), 29 March 2019, in Florence, Italy. https://osda.gitlab.io

these devices are often only used in low volume, specific use cases where they are not exposed to the more extreme use and abuse which commercial devices tend to see. Hence the community has not faced the same driving forces which necessitate investment in verification infrastructure.

A similar point can be made for academic or research projects which produce OSH artifacts. The unfortunate set of motivations in academia can mean little or no incentive is present for researchers to invest the time in verifying their designs. This means the community suffers in terms of design re-use, and the trustworthiness of any reported experimental results. There is also a human difficulty in interesting students and researchers in verification generally, which can be seen as the somewhat less glamorous aspect of a project. The current European Union mandate for "open access" research makes it even more important that academic hardware artifacts show evidence of verification effort.

While there are several notable and worthy exceptions to this state of affairs, we maintain that these are indeed exceptions, and that generally OSH verification capability and practice lags behind what is seen in industry.

This is in stark contrast to the OS *software* community; where there is considerable contribution to OS projects from industry, as well as dependence on OS software by industry. Because of this, OS software tooling and development practices have been able to advance in concert with commercial counterparts. Indeed, many software companies make a virtue of contribution to open source software projects. The same cannot be said in the context of the traditional EDA / hardware IP companies.

B. Approaches to hardware verification

Here we briefly describe several approaches and aspects to hardware verification used in industry.

1) Constrained Random Verification: Constrained Random Verification (CRV) techniques centre around automatically generating random input stimulus, which is constrained to only useful or interesting parts of the input domain of the design. For example, one may test a CPU by randomly feeding it 32-bit instruction words, though this might involve much wasted effort unless the randomised words are constrained to be mostly valid instructions.

CRV has seen much collaborative development across industry, first through the Open Verification Methodology (OVM) standard, and now the Universal Verification Methodology (UVM) standard [4]. EDA vendor specific tools such as the E language from Cadence also support CRV, but are not so widely used as they are not vendor neutral.

It is important to note that UVM is not just a library of code, but also a prescriptive set of methods for building testbenches. This includes how to partition code into re-usable blocks, and a standard pattern for arranging these blocks within a testbench architecture. Such methods have made it extremely easy for engineers across different projects or even companies to quickly understand and use a new testbench, focusing on modelling the actual design rather than picking apart the testbench hierarchy.

2) Coverage Metrics: One must be confident that all different areas of a design are actually stimulated by the testbench. For this, coverage metrics are used.

There are several different kinds of coverage metric used in hardware verification which have direct analogues in the software world. Line coverage of HDL source files, as well as condition coverage (branch coverage in software) are easily collected by simulation tools.

Additionally to this, hardware verification uses two notions of coverage not often found in software development. The first is toggle coverage, which records how many times each signal has transitioned from 0 to 1 and 1 to 0. The second and perhaps most powerful is functional coverage.

Functional coverage is much more abstract than the other metrics described. It is designed to capture whether or not higher level behaviour in a design has been observed during simulation. For example, to say that an add instruction has "been executed", it is not enough to simply know we have fed the design an add instruction. One must also record that it was decoded correctly, tracked through the pipeline, produced the correct side effects and retired successfully. All of these observed together count as the functional covering of the add instruction. This notion would then be extended to see that we execute all or as many variants as is feasible of the add instruction with different operands and input values. As with stimulus generation, it is possible to make functional coverage models re-usable across different designs. This is particularly useful for CPU ISAs, where there may be many different implementations of the same specification.

Coverage metrics are often developed in parallel to the stimulus generation parts of a testbench. By recording what the design actually does using coverage metrics, one can identify gaps in stimulus generation and fill them.

3) Formal Verification: In the context of hardware verification, this refers to use of Bounded Model Checking (BMC), Satisfiability Modulo Theories (SMT) and other formal reasoning methods to prove that some set of correctness properties about the design hold true. Despite having existed as a technique for hardware verification for some decades, it has only recently begun to be used regularly and on similar design sizes as traditional CRV techniques.

The main advantages of formal methods in verification is that they cover the entirety of the input space. This means that very subtle bugs can be found very quickly, which might take much longer with traditional simulation based approaches. Such methods consequently give an extremely high degree of confidence about the design, which is essential for high-assurance applications like automotive CPUs.

While CRV is a very general method applicable to most designs, formal verification requires more care in how and where it is applied. It can suffer from state-space explosion, where the memory required to represent the possible states of the design make it infeasible to work with. This means that formal verification tends to lend itself better to verification of

This work was presented at the first Workshop on Open-Source Design Automation (OSDA), 29 March 2019, in Florence, Italy. https://osda.gitlab.io

control paths and logic, rather than data paths. With careful management however, there is no reason formal methods cannot be applied to data-paths.

Unlike CRV (more specifically, UVM based CRV), there is not the same level of industry standard methods for managing a formal verification effort. However, several papers exist [5], [6], [7], [8], [9], [10] which give good overviews of how to structure a formal verification flow, and there are enough similarities between them to suggest that consciously or not, there is an ongoing trend toward a common method.

4) Trends in functional verification: Here we extract some key points from the Wilson Research Group Functional Verification Studys run on behalf of Mentor Graphics [3], [11], [12]. The 2018 study in focuses on FPGA verification in particular, which makes it more relevant to the OSH community given the better tooling availability than for ASICs. Some of the key findings from the reports relevant to the OS community are:

FPGA designs are including more embedded CPU cores [12]. This implies a need for robust reusable verification tools for the cores themselves, and ways to verify their integration into a wider system.

FPGA and ASIC systems are seeing a transition from block-level verification to system-level verification. As more OS components appear, integrating them correctly into a complete system is an inevitable challenge for the community.

More designs are including dedicated security hardware [12]. These components require particular attention from a verification perspective, as their system level interactions can reveal side-channels for attack. Making verifiable claims as to the security of OSH and systems will become increasingly important if the promise of auditable hardware is to be realised. Section III-D discusses security challenges in more detail.

As of 2018, 84% of surveyed FPGA design projects suffered a "non-trivial bug escape into production". The trends for causes of bug are particularly illustrative. While functional / logic bugs dominate (underlining the need for effective block-level verification), the percentage of these is decreasing over time. According to [12], "this reduction of 'logic and functional flaws' is likely due to the FPGA market maturing its verification processes" and "increased adoption of mature design IP for integration". The implication for OSH is that re-usable verification IP is a worthy investment. Functional and logic flaws have been replaced with problems in timing, clocking, crosstalk and firmware. Firmware is expected, given the corresponding increase in embedded CPU usage.

We suggest that there is considerable scope for the OS community to learn from historic verification trends in industry. As a community, we are uniquely forwarned in terms of the verification challenges yet to be faced, and can make more informed decisions about investments in tooling as a result.

C. Open source and industrial tool comparisons

From Table I, we see even though the table is far from exhaustive in its tool listing, most coarse grain features of a hardware implementation flow are already supported by OS tools. However, the degree and flexibility of that support often lags that of the commercial tool-sets. Further, as is the case with verification, there is a difference between tool capability, and exercising of that capability by OS projects.

D. Case Study: OpenCores.org

Here we conduct a very simple survey of the OpenCores website[13], a popular repository for OSH designs. As of December 2018, OpenCores contains 1182 projects, where each project represents some hardware IP block.

Of these, 31 (2.6%) are "OpenCores Certified". This means the project meets some completeness requirements. These include presence of the design files, flow scripts, documentation and "self-checking testbenches".

OpenCores also labels projects based on their status, which ranges from "planning" through "alpha", "beta" and "stable" to "mature". To qualify as a "stable" or "mature" project, the design must be tested and validated by the project developers and community members. There are 47 (4%) mature projects and 475 (40%) stable projects.

Across the site, there is no standard way of *quantifying* the level of confidence in the functional correctness of a design, or reporting coverage numbers. This is an essential set of metrics for any repository of IPs. Without them, it is impossible to be confident in a design without personally checking it. In the software world, project hosting sites like Github, Gitlab and 3rd party plugin providers make it very easy to show up front the testing and coverage status of a project. We suggest this practice be adopted by the OSH community without delay.

Further, there is a heavy skew on the site toward hardware components rather than re-usable verification IP. Only 33 (2.8%) of the listed projects are categorised under "Testing / Verification". We believe this further underlines a need for the OSH community to invest more in re-usable verification infrastructure, verification efforts generally, and reporting / quantifying the verification effort that has been undertaken.

III. Challenges in open source hardware verification

A. Behaviour Specification

One of the hardest problems in both hardware and software development is the actual specification of intended behaviour. Historically, this has been done using natural language specifications. However this leaves considerable room for misinterpretation and ambiguity.

More recently, there have been efforts to create formalised specifications, from which large amounts of both design and verification infrastructure can be generated or mechanically derived. The benefits from such approaches are numerous, and analogues exist in other fields showing how powerful formalised specifications are. One example is machine-checked proofs in cryptography, where a formalised mathematical expression of a cipher or protocol is used to prove its security under some adversarial model. As [14] shows, these types of formal specification are directly relevant to hardware design as well.

This work was presented at the first Workshop on Open-Source Design Automation (OSDA), 29 March 2019, in Florence, Italy. https://osda.gitlab.io

Tool	Coverage Collection	Coverage Management	Simulation	Formal Proofs	Waveform Analysis	Synthesis	Place Route	Open Source
Verilator	X		X					X
Icarus Verilog			X					X
Yosys				X*		X		X
GTKWave					X			X
GHDL			X					X
Qflow							X	X
nextpnr							X	X
Verilog To Routing							X	X
Synopsys VCS	X		X					
Synopsys VC Formal				X				
Synopsys Verdi		X			X			
Synopsys DC						X		
Synopsys ICC							X	

TABLE I

A COARSE FEATURE COMPARISON BETWEEN VARIOUS OS TOOLS AND (FOR THE SAKE OF EXAMPLE) THE COMMERCIAL SYNOPSYS TOOL SET. NOTE: YOSYS ITSELF DOES NOT NECESSARILY PERFORM THE FORMAL PROOFS, BUT CONVERTS HDL INTO A FORM SUITABLE FOR PERFORMING FORMAL PROOFS ON IT.

In industry, the recent introduction of the Portable Test and Stimulus Standard (PSS) [15] by the Accellera standards body goes some way to addressing this problem. There are various other commercial and OS projects attempting to create formalised specifications of behaviour [16], [17], [18], [6]. Particularly noteworthy is the effort to create a formalised specification for the RISC-V ISA [19], [20], [21], [22]. While the development process for this is not open, many of the candidates are, and provide informative examples how to create such specifications.

For OSH projects, accurate specification of behaviour will become more critical as the complexity of designs and the number of functioning agents within the system increase. While we make no recommendations as to which method of formal specification should be used, there are considerable advantages to the OS community in adopting their use more generally.

Specific challenges to adopting such techniques will revolve around correctness preserving transformation and mapping of formalised specifications onto the actual designs, and verifying that one matches the other.

B. Formal Verification

Thanks to the SMT2[23] backend of Yosys[24], the OS community is already well placed use formal methods for verifying designs. There are already some impressive examples of re-usable formal verification frameworks for RISC-V CPUs developed using Yosys [17].

In terms of future challenges, there is considerable scope for developing standard design patterns for formal verification tools. There are a number of industrial papers on structuring formal verification efforts, and identifying designs most amenable to formal methods [10], [8], [5]. Most of the techniques they describe are transferable into existing OS tools, what is lacking are easily accessible and explicit "how to" guides for how to apply them to larger designs.

One missing piece of functionality at the moment is the notion of "property set completeness", as described in [5]. This is an automatic way of checking the set of correctness properties being used cover enough of the design. Without this automated check, one must manually audit the set of correctness properties to ensure all parts of the design functionality are hit.

Given intended design functionality is usually expressed much more abstractly than the actual design itself (often natural language vs. RTL), this also leads to significant effort being required to correctly abstract the right information at the right time from the design. An incorrect abstraction will render any formal verification results useless. While careful structuring of the interface to formal checkers can help manage this, there is scope for developing ways to correctly and robustly map high level specifications of intended design behaviour onto lower level implementations of those specifications.

We suggest there is also scope for extending existing formal methods to work at a system level. Given the number of bugs which manifest due to either integration errors or interactions between hardware and software [12], as well as the security claims made at a system level (see section III-D), it is not enough to show that individual blocks are functionally correct. As the OSH community moves from developing block level designs to system level designs, verifying integration of many different blocks will become increasingly important.

C. CRV and Coverage Metrics

CRV and coverage metrics are areas where there is a large capability and practice gap between the OS and industrial communities. Part of this is due to UVM being implemented in SystemVerilog. SystemVerilog is an enormously complicated language for which only commercial tool support exists. While it may not be a sensible use of time to build an OS implementation of SystemVerilog; replicating some of its language features such as functional coverage specification and collection would be a useful contribution to existing tools.

In terms of stimulus generation, we believe it is worthwhile to take many of the method and design patterns described in UVM, and recreate them for OS usage. Open source examples of re-creating the UVM approach include Coverify's vlang implementation of UVM [25] and UVVM for VHDL

This work was presented at the first Workshop on Open-Source Design Automation (OSDA), 29 March 2019, in Florence, Italy. https://osda.gitlab.io

[26]. Likewise, AMD and Cadence have collaborated on a multi-language framework for UVM, called UVM-ML [27]. Several new hardware design languages (Clash [28], Chisel [29], MyHDL [30], SpinalHDL [31]) reference how their host language can make stimulus and re-usable verification infrastructure easier to express. However, we see little evidence of this in existing OS projects. A concerted effort to build stimulus generation frameworks in these languages, which are interoperable and/or usable with existing Verilog or VHDL designs would be very valuable to the entire community.

The same is true for coverage collection. Line, condition and toggle coverage should become the bare minimum of reported statistics for projects claiming any kind of verification effort. We note that this is not strictly relevant for projects relying totally on formal verification methods, in which case things like environment assumptions should be stated.

Again, there is considerable scope for developing ways to express re-usable functional coverage metrics that can be integrated with existing simulation tools. This is especially important given the limited support for functional coverage modelling with existing OS HDL simulators. We note Verilator supports specification of functional coverage using SystemVerilog Assertion (SVA) syntax, but not the actual SystemVerilog Coverage features. Without clear reporting of coverage metrics, it is impossible to quantify the degree of confidence in any constrained random verification effort.

D. Security Claims and Assurances

A common argument for adopting OSH is its auditability, and the confidence this gives in it's security claims over a closed source counterpart.

While this is true in generality, it is a point derived partially from the software world, and some important distinctions apply. Open source software security claims are predicated on the assumption vulnerable software is updated when a bug is found. This is inevitably more difficult for hardware. The picture is further complicated by how some vulnerabilities only manifest at the system level, due to unanticipated interactions between otherwise secure or trustworthy components.

To make OSH truly more trustworthy than closed source counterparts, considerable effort will be required to audit whole systems prior to their deployment. This is arguably where OSH has an opportunity to do better than commercial hardware, since such audits can be undertaken publicly, and by many more people. Indeed, this is particularly important, since hardware security is a specialist field with relatively few practitioners. The easier it is for this knowledge to be shared and put to use in the open, the more confidence we can have in OSH security. Further, all security claims are based on the assumption the underlying system and components are functionally correct. Thus, any security claims must be backed up by evidence of functional verification.

E. Human Factors

While technical challenges to enabling better verification of OS hardware described above are important, in our experience there are several human and organisational factors which will play a more influential role. In terms of methodology and how to organise development of OS hardware, there are several key differences to commercial development and culture which must be faced.

In industry, development teams are often rigidly structured in terms of their responsibilities. Indeed, verification and design teams are often kept disjoint as a safeguard against misinterpreting specifications. As a result, many development methods in industry do not directly transfer into the kind of OS community we see in the software world. This world is typically more disparate, with individuals responsible for all aspects of parts of a code base. It is not reasonable to expect the OS community to rigidly organise itself to practice commercial development methods. Nonetheless, as more and larger OSH projects appear, this will become an organisational challenge to be aware of, and we expect to see new ways of organising hardware development to appear which have not had the chance to flourish in commercial development environments.

A related point is how many OS software enthusiasts are becoming part of the OSH community. While this is heartening and certainly a positive development, there are cautions to be mindful of as well. The software development mindset has become dominated by "agile" development practices [32]. While software has the luxury of being continually updatable, for many hardware designs (ASIC or FPGA) there is often a hard stop to development when the design is finally deployed. The mindset of "it can be fixed in a later release" or the concept of a "minimum viable product" rarely apply to hardware development. This does not preclude the adoption of agile practices by the hardware community, and there has already been discussion about how and whether to adopt agile methods in the context of hardware development [33], [34]. While answers to these questions are beyond the scope of this paper, the mix of development methodologies from software and hardware worlds which are bound to meet in the OSH community is an important consideration going forward.

Given the mix of technical backgrounds people in the OSH community will likely have, we believe one of the key intersecting challenges will be the *attitude* to design verification. Rather than focusing on enabling OSH *design*, we suggest OSH *development* may be a more appropriate term. While the distinction is small, it should be recognised that design methodology is but one aspect of development and that verification strategy is also a vital component. Indeed, we should foster a mindset which emphasises that verification is a non-optional part of any OSH project.

IV. Conclusion

The OSH community has an excellent amount of momentum behind it in terms of enabling larger and more complex designs. There is a danger however, that the verification and methodological aspects of hardware development are being neglected. If OSH is to achieve the same kind of impact as OS software, more emphasis must be placed on evidenced

This work was presented at the first Workshop on Open-Source Design Automation (OSDA), 29 March 2019, in Florence, Italy. https://osda.gitlab.io

verification efforts. We make the following concrete suggestions as to how projects and individuals within them can start combating this:

Open source hardware projects should state how much functional verification has been undertaken. This should be qualified by function/line/condition/toggle coverage numbers for CRV based approaches, and/or an explanation of the formal properties proved as appropriate. At a bare minimum, these coverage numbers should be immediately obvious to anyone considering re-using an existing design.

Any verification infrastructure development should be done with reusability in mind. Projects like OpenCores[13] serve as a good example of a repository of re-usable components, but presently is heavily weighted in favour of hardware designs, rather than reusable verification infrastructure. This is particularly useful and achievable for things like assertions for bus interfaces and models of CPU ISAs. One should not need to re-write a set of checkers for an AXI / Wishbone interface for every project.

The same goes for stimulus generation. There is considerable scope for contribution around building constrained random stimulus generators for the newer high-level HDLs, or even for Verilog via its VPI interface.

As well as re-usable technical artifacts, effort should be made to share within the community successful approaches to hardware verification, and to adopt/adapt best practice from industry. This is particularly true with formal verification approaches, where the OS community has already had considerable success in re-usable infrastructure.

In industry, there is a large volume of training material available for the various verification tools and methods. Curating a similar set of "how to" guides and tutorials focused on OS tooling would be a valuable contribution to the community.

As a community, we must make verification and accurate reporting of verification effort a first class component of OSH development.

References

[1] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin *et al.*, "Meltdown: Reading kernel memory from user space," in *27th {USENIX} Security Symposium ({USENIX} Security 18)*, 2018, pp. 973–990.

[2] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," *arXiv preprint arXiv:1801.01203*, 2018.

[3] H. D. Foster, "Trends in functional verification: a 2014 industry study," in *Proceedings of the 52nd Annual Design Automation Conference*. ACM, 2015, p. 48.

[4] Accellera, "Uvm (universal verification methodology)," https://www.accellera.org/downloads/standards/uvm.

[5] J. Bormann, S. Beyer, A. Maggiore, M. Siegel, S. Skalberg, T. Blackmore, and F. Bruno, "Complete formal verification of tricore2 and other processors," in *Design and Verification Conference (DVCon)*, 2007.

[6] U. Kühne, S. Beyer, J. Bormann, and J. Barstow, "Automated formal verification of processors based on architectural models," in *Formal Methods in Computer-Aided Design (FMCAD), 2010*. IEEE, 2010, pp. 129–136.

[7] R. Baranowski and M. Trunzer, "Complete formal verification of a family of automotive dsps," *DVCon Europe*, 2016.

[8] I. Beer, S. Ben-David, D. Geist, R. Gewirtzman, and M. Yoeli, "Methodology and system for practical formal verification of reactive hardware," in *Computer Aided Verification*, D. L. Dill, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1994, pp. 182–193.

[9] N. Kim, J. Park, H. Singh, and V. Singhal, "Sign-off with bounded formal verification proofs," in *Design and Verification Conference (DVCon)*, 2014.

[10] H. Foster, L. Loh, B. Rabii, and V. Singhal, "Guidelines for creating a formal verification testplan," *Proc. DVCon*, 2006.

[11] H. D. Foster, "Trends in functional verification: a 2016 industry study," 2016.

[12] ——, "Trends in functional verification: a 2018 industry study," 2018.

[13] Various Authors, "Opencores.org," https://opencores.org.

[14] J. R. Kiniry, D. M. Zimmerman, R. Dockins, and R. Nikhil, "A formally verified cryptographic extension to a risc-v processor," 2018.

[15] Accellera, "Pss 1.0 language reference manual," https://www.accellera.org/downloads/standards/portable-stimulus.

[16] "End-to-End Verification of Arm Processors with Isa-Formal," in *Proceedings of the 2016 International Conference on Computer Aided Verification (CAV16)*, ser. Lecture Notes in Computer Science, S. Chaudhuri and A. Farzan, Eds., vol. 9780, no. 9780. Springer Verlag, July 2016, pp. 42–58.

[17] C. Wolf, "End-to-end formal isa verification of risc-v processors with riscv-formal," http://www.clifford.at/papers/2017/riscv-formal/slides.pdf.

[18] J. Choi, M. Vijayaraghavan, B. Sherman, A. Chlipala *et al.*, "Kami: a platform for high-level parametric hardware specification and its modular verification," *Proceedings of the ACM on Programming Languages*, vol. 1, no. ICFP, p. 24, 2017.

[19] T. Bourgeat, "Formal semantics of riscv," https://content.riscv.org/wp-content/uploads/2018/05/slidesThomasBourgeat.pdf.

[20] R. Nikhi, "A formal spec of the risc-v instruction set architecture, written in bluespec bsv," https://github.com/rsnikhil/RISCV_ISA_Formal_Spec_in_BSV.

[21] A. Armstrong, T. Bauereiss, B. Campbell, S. Flur, K. E. Gray, P. Mundkur, R. M. Norton, C. Pulte, A. Reid, P. Sewell *et al.*, "Detailed models of instruction set architectures: From pseudocode to formal semantics," in *25th Automated Reasoning Workshop*, 2018, p. 23.

[22] T. Bourgeat, "Formal semantics of risc-v," https://content.riscv.org/wp-content/uploads/2018/05/slidesThomasBourgeat.pdf.

[23] C. Barrett, A. Stump, and C. Tinelli, "The SMT-LIB Standard: Version 2.0," in *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, UK)*, A. Gupta and D. Kroening, Eds., 2010.

[24] C. Wolf, "Yosys open synthesis suite," http://www.clifford.at/yosys/.

[25] Coverify, "Vlang port of uvm (universal verification methodology)," https://github.com/coverify/vlang-uvm.

[26] Bitvis, "Open Source VHDL Verification Library and Methodology," https://github.com/UVVM/UVVM.

[27] Advanced Micro Devices and Cadence Design Systems, "UVM-ML Open Architecture version 1.10.2," http://forums.accellera.org/files/file/65-uvm-ml-open-architecture/.

[28] C. Baaij, M. Kooijman, J. Kuper, A. Boeijink, and M. Gerards, "Clash: Structural descriptions of synchronous hardware using haskell," in *Digital System Design: Architectures, Methods and Tools (DSD), 2010 13th Euromicro Conference on*. IEEE, 2010, pp. 714–721.

[29] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, "Chisel: constructing hardware in a scala embedded language," in *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*. IEEE, 2012, pp. 1212–1221.

[30] J. Villar, J. Juan, M. Bellido, J. Viejo, D. Guerrero, and J. Decaluwe, "Python as a hardware description language: A case study," in *Programmable Logic (SPL), 2011 VII Southern Conference on*. IEEE, 2011, pp. 117–122.

[31] Various Authors, "Spinalhdl," https://github.com/SpinalHDL/SpinalHDL.

[32] K. Beck, M. Beedle, A. Van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries *et al.*, "Manifesto for agile software development," 2001.

[33] M. Bartlet, "Is agile coming to hardware development?" https://www.design-reuse.com/articles/37187/is-agile-coming-to-hardware-development.html.

[34] N. Johnson, "IC Development and the Agile Manifesto," http://agilesoc.com/articles/ic-development-and-the-agile-manifesto/.

This work was presented at the first Workshop on Open-Source Design Automation (OSDA), 29 March 2019, in Florence, Italy. https://osda.gitlab.io