AFRL-RY-WP-TR-2023-0221

UPSCALE: SCALING UP FORMAL TOOLS FOR POSH OPEN SOURCE HARDWARE

Clark Barrett
Stanford University

OCTOBER 2023
Final Report

AIR FORCE RESEARCH LABORATORY
SENSORS DIRECTORATE
WRIGHT-PATTERSON AIR FORCE BASE, OH 45433-7320
AIR FORCE MATERIEL COMMAND
UNITED STATES AIR FORCE

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with The Under Secretary of Defense memorandum dated 24 May 2010 and AFRL/DSO policy clarification email dated 13 January 2020. This report is available to the general public, including foreign nationals.

Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RY-WP-TR-2023-0221 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

//Signature//
CHRISTOPHER A. BOZADA
Program Manager
Aerospace Components and Subsystems Division

//Signature//
GENE M. WILKINS, Lt Col, USAF
Deputy Chief
Aerospace Components and Subsystems Division
Sensors Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

*Disseminated copies will show "//Signature//" stamped or typed above the signature blocks.

REPORT DOCUMENTATION PAGE

PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ORGANIZATION.

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED	
		START DATE	END DATE
October 2023	Final	19 July 2018	31 December 2022

4. TITLE AND SUBTITLE

UPSCALE: SCALING UP FORMAL TOOLS FOR POSH OPEN SOURCE HARDWARE

5a. CONTRACT NUMBER	5b. GRANT NUMBER	5c. PROGRAM ELEMENT NUMBER
FA8650-18-2-7854	N/A	N/A

5d. PROJECT NUMBER	5e. TASK NUMBER	5f. WORK UNIT NUMBER
N/A	N/A	Y1TJ

6. AUTHOR(S)

Clark Barrett

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Stanford University
424 Santa Teresa St,
Stanford, CA 94305

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)		10. SPONSOR/MONITOR'S ACRONYM(S)	11. SPONSOR/MONITOR'SREPORT NUMBER(S)
Air Force Research Laboratory Sensors Directorate Wright-Patterson Air Force Base, OH 45433-7320 Air Force Materiel Command United States Air Forces	Defense Advanced Research Projects Agency (DARPA/MTO) 675 North Randolph Street Arlington, VA 22203	AFRL/RYD	AFRL-RY-WP-TR-2023-0221

12. DISTRIBUTION/AVAILABILITY STATEMENT

DISTRIBUTION STATEMENT A. Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES

This material is based on research sponsored by the Air Force Research Laboratory (AFRL) and the Defense Advanced Research Projects Agency (DARPA). The U.S. Government (USG) is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the AFRL, DARPA, or the USG. Report contains color.

14. ABSTRACT

The POSH Upscale Project developed tools and techniques for verifying and evaluating open-source hardware. Modern Systems-on-Chip (SoCs) pose two distinct challenges to this endeavor. The first is increasing heterogeneity. SoCs comprise a range of programmable processors, dedicated hardware function blocks, and analog/mixed-signal (AMS) components. The second is differing levels of abstraction used across these components in verification tools—instruction-level hardware-software interfaces, register-transfer level finite state machine models, and circuit-level models.

15. SUBJECT TERMS

System-on-chip, open-source hardware, circuit verification, analog/mixed-signal, circuit abstraction

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES
a. REPORT	b. ABSTRACT	C. THIS PAGE		
Unclassified	Unclassified	Unclassified	SAR	23

19a. NAME OF RESPONSIBLE PERSON	19b. PHONE NUMBER *(Include area code)*
Christopher Bozada	N/A

Table of Contents

Section **Page**

List of Figures

Figure **…Page**

1 UPSCALE: SCALING UP FORMAL TOOLS FOR POSH OPEN SOURCE HARDWARE

Modern society relies increasingly on electronics of every kind. As electronics multiply and diversify, there is a growing need to build new hardware components quickly and correctly. The POSH program aimed to seed an open-source library of hardware modules, with the goal of producing a thriving open-source hardware community like what already exists for open-source software.

The goal of the POSH Upscale Project (Fig. 1) was to develop tools and techniques for verifying and evaluating open-source hardware. Modern Systems-on-Chip (SoCs) pose two distinct challenges to this endeavor. The first is increasing heterogeneity. SoCs comprise a range of programmable processors (CPUs, GPUs, other domain specific processors), dedicated hardware function blocks (accelerators, memory controllers, on-chip networks, and buses), and analog/mixed-signal (AMS) components. The second is differing levels of abstraction used across these components in verification tools—instruction-level hardware-software interfaces for processors, register-transfer level finite state machine models for other digital modules, and circuit-level models for AMS components.

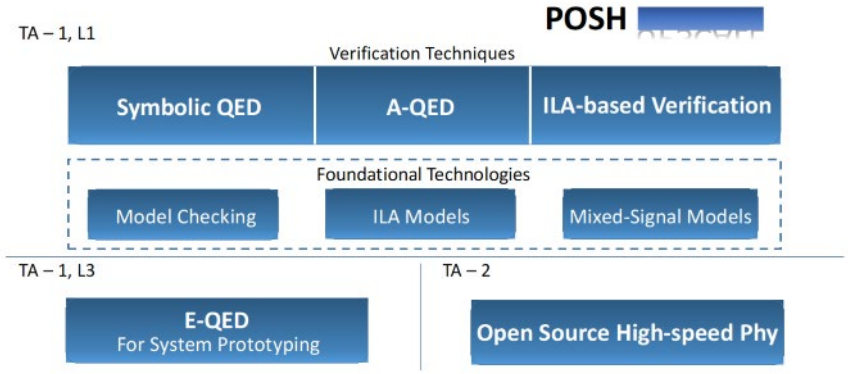

Figure 1: The POSH Upscale Project

The Upscale project addressed both of these challenges and the additional challenge of providing open access by developing and building on three foundational technologies: (i) open-source model checkers as an alternative to commercial tools for performing formal verification queries (see Sec. A); (ii) instruction- level abstractions (ILAs) that provide a layer of abstraction similar to that provided by instruction set architectures, but for arbitrary hardware modules (see Sec. B); and (iii) mixed-signal models needed to allow formal techniques to be applied to designs with analog components (see Sec. C).

Based on these fundamental technologies, we explored three different formal verification techniques (targeting the formal analysis sub-task of TA-1), with complementary strengths to address heterogeneous IP blocks, designed to maximize both automation and applicability. The

first was Symbolic QED, a fully automatic methodology for verifying processor cores or SoC's with at least one processor core (see Sec. D). Symbolic QED depends on model checking and (for SoC's with analog components) mixed- signal models. The second was A-QED, a set of techniques (from mostly to fully automatic) for stand- alone verification of accelerators (see Sec. D). A-QED requires model checking as well as ILA models. A-QED may also require mixed-signal models if analog components are present in an accelerator. The third was ILA-based verification. ILA-based verification is a very general approach, supporting varying degrees of automation. Moreover, it can often be used when other techniques fail or are not applicable (see Sec. B). ILA-based verification requires model checking and ILA models, and may additionally require mixed-signal models to deal with modules with analog components.

In addition to these efforts, we supported the broader POSH objectives with two additional activities:
(i) exploring a system prototyping tool-set (see Sec. D); and building an open-source high-speed phy component (a TA-2 task) as a proof of concept of the techniques developed for mixed-signal modeling (see Sec. C). In the rest of the report, we detail the advances and accomplishments in each of the areas mentioned above.

1.1 Model Checking

Model checking [16, 40] is an influential verification capability in modern system design. Its greatest success has been with finite-state systems, where propositional methods such as binary decision diagrams (BDDs) [9] and Boolean satisfiability (SAT) solvers [46] are used as verification engines. At the same time, significant efforts have been made to lift model checking techniques from finite-state to infinite-state systems [7, 12, 13, 15, 20, 42]. This requires more expressive verification engines, such as solvers for satisfiability modulo theories (SMT) [5].

As part of the Upscale effort, we developed a new word-level SMT-based model checker, Pono [37].

Pono was designed with three use cases in mind: 1) *push-button verification*; 2) *expert verification*; 3) *model checker development*. For 1, Pono provides competitive implementations of standard model checking algorithms. For 2, it exposes a flexible API, affording expert users fine-grained control over the tool. This can be useful in traditional model checking tasks (e.g., manually guiding the tool to an invariant, or adjusting the encoding for better performance), but it also enables the tool to be easily adapted for other tasks. In addition, Pono is designed using a completely generic SMT solver interface called smt- switch [38], making it trivial to experiment with different back-end solvers. For 3, Pono is open source [3] and designed to be easily modifiable and extensible with a simple, modular, and hierarchical architecture. Taken together, these features make it relatively easy to do controlled experiments by comparing results obtained using Pono, while varying only the SMT solver or the model checking algorithm.

Pono has been used in a variety of research projects, both for model checking and other custom applications. It has also been used in two graduate level courses at Stanford University, where students used both the command-line interface and the API. With this promising start, we hope it will have a long and productive existence supporting research, education, and industry.

1.1.1 Capabilities

In this section, we highlight some key capabilities of Pono. The design makes use of abstract interfaces and inheritance to make it easy to add or extend functionality. Base class implementations of core functionality are provided but are kept simple to prioritize readability and transparency. And, of course, they can be overridden using inheritance and virtual functions.

We start by describing the engines provided for push-button verification. Next, we take a closer look at two ways that the basic architecture can be extended.

Main Engines. Pono has several engines, all of which have been lifted to the SMT-level. The main engines include:

- Bounded Model Checking [6] (88 LoC);
    - K-Induction [45] (161 LoC);
    - Interpolant-based Model Checking [41] (230 LoC);
    - IC3-style algorithms [8] (see below for LoC).

The engines leverage reusable infrastructure for manipulating circuit representations.

IC3 Variants. IC3 is widely recognized as one of the best-performing algorithms for SAT-based model checking [18]. Liftings to SMT are an area of active research and have produced several variations with promising results [7, 15, 30]. To support this active research direction, Pono includes a special IC3 base class, which implements a framework common to all variations of the algorithm.[1] Current instantiations of IC3 implemented in Pono include:

[1]For details on how the IC3 algorithm works, we refer the reader to [8, 18].

- IC3: a standard Boolean IC3 implementation [8, 18] (152 LoC);
- IC3Bits: a simple extension of IC3 to bit-vectors, which learns clauses over the individual bits (113 LoC);
- Model-based IC3: a naive implementation of IC3 lifted to SMT, which learns clauses of equalities between variables and model values (397 LoC);
- IC3IA: IC3 via Implicit Predicate Abstraction [15] (456 LoC);
- IC3SA: a basic implementation of IC3 with Syntax-Guided Abstraction for hardware verification [21] (984 LoC);
- SyGuS-PDR: a syntax-guided synthesis approach for inductive generalization targeting hardware designs [58] (1047 LoC).

Counterexample-Guided Abstraction Refinement (CEGAR). CEGAR [33] is a popular framework for iteratively solving difficult model checking problems. It is typically parameterized by the underlying model checking algorithm, which operates on an abstract system that is iteratively refined as needed. Pono provides a generic CEGAR capability. We describe two example uses of the CEGAR infrastructure implemented in Pono.

Operator Abstraction. This simple CEGAR algorithm uses uninterpreted functions (UF) to abstract potentially expensive theory operators (e.g. multiplication). The implementation is parameterized by the set of operators to replace with UFs. The refinement step analyzes a counterexample trace by restoring the concrete theory operator semantics. If the trace is found to be spurious, constraints are added to enforce the real semantics for the abstracted operators (e.g., equalities between certain abstract UFs and their theory operator counterparts), thus ruling out the spurious counterexample.

Counterexample-Guided Prophecy. This CEGAR approach replaces array variables with initially memoryless variables of uninterpreted sort and replaces the `select` and `store` array operators with UFs [36]. Due to the array theory semantics, it is not always possible to remove spurious counterexamples with quantifier-free refinement axioms over existing variables. However, instead of using potentially expensive quantifiers, the algorithm adds auxiliary variables (history and prophecy variables) [4], which can rule out spurious counterexamples of a given finite length. This approach has the effect of removing the need for array solving and can sometimes prove properties using prophecy variables that would otherwise require a universally quantified invariant.

1.1.2 Performance

We evaluated Pono on the 2020 Hardware Model Checking Competition (HWMCC) benchmarks. The benchmarks are split into bitvector-only and bitvector plus array categories. We evaluate against AVR [22, 1] and CoSA2 [2] (a previous name and version of Pono), the winners of HWMCC 2020 and HWMCC 2019, respectively. We also compare against sygus-apdr (the reference implementation of SyGuS- PDR [58]) on the bitvector benchmarks (as sygus-apdr targets bitvectors). We ran all 16 configura- tions of AVR from their HWMCC 2020 entry: several configurations of BMC and k-induction, and 11 configurations of IC3SA. We ran the 4 configurations of CoSA2 from the HWMCC 2019 entry: two BMC configurations, k-induction, and interpolant-based model checking. We ran sygus-apdr with 4 different parameters controlling the grammar for lemmas. For the bitvector-only benchmarks, we ran Pono with 10 configurations: 3 configurations of IC3IA, 2 configurations of IC3SA, 2 configurations of SyGuS-PDR, IC3Bits, k-induction, and BMC. For the array benchmarks, we ran 5 configurations: 3 configurations of IC3IA (one with Counterexample-Guided Prophecy), k-induction, and BMC. We show our results on the HWMCC 2020 benchmarks in Fig. 2. AVR wins in both categories, although Pono is fairly competitive, outperforming the other tools.

result	BV (324 total)				BV + Array (315 total)		
	Pono	AVR	CoSA2	sygus-apdr	Pono	AVR	CoSA2
safe	183 (283s)	**215** (115s)	98 (283s)	115 (545s)	252 (224s)	**274** (63s)	209 (299s)
unsafe	47 (314s)	47 (220s)	41 (232s)	15 (279s)	19 (208s)	19 (352s)	19 (204s)
total	230 (289s)	**262** (134s)	139 (268s)	130 (514s)	271 (223s)	**293** (82s)	228 (291s)

Figure 2: Results on HWMCC2020 Benchmarks

These results show that Pono is well on its way to being both widely applicable and performance- competitive. The arithmetic experiments demonstrate the capabilities of its IC3IA engine, but other engines have some room for improvement. Both IC3SA and SyGuS-PDR were recently added to Pono, and its implementation of these algorithms still lags the corresponding implementations in AVR and sygus-apdr, respectively. There are also some features that are known to help performance and are not yet implemented in Pono. For example, the best configurations of AVR use UF data abstraction. This differs from our UF operator abstraction in that it replaces all abstracted data with uninterpreted sorts and learns targeted data refinement axioms.

1.2 Instruction-Level Abstraction

In work that was ongoing when the POSH project started, we had proposed a uniform and formal abstraction for processors and accelerators that captures their software-visible functionality. This abstraction is called an Instruction-Level Abstraction (ILA). It is based on the familiar notion of computation triggered by "instructions." For a processor, the ILA is based on the ISA (Instruction Set Architecture). For an accelerator, the insight is that commands at its interface are akin to instructions in a processor. Thus, just as the ISA models processor behavior through specifying state changes resulting from each instruction, the ILA models accelerator behavior by specifying state changes resulting from each of its "instructions," i.e., its commands. Further, as with ISAs, this modeling can distinguish the state that is persistent be- tween instructions (architectural state), from implementation state (micro-architectural state). Top-down this modeling provides a specification for functional verification of hardware, and bottom-up it provides an abstraction with operational semantics for software/hardware co-verification.

Since abstraction is crucial for formal analysis of complex designs, Upscale leveraged and built upon ILAs to create stronger validation of POSH blocks. Accordingly, Upscale research objectives comprised three main thrusts: (i) development of an ILA modeling language, (ii) ILA-based specification, and (iii) ILA-based verification. The results from these thrusts are described in detail below.

1.2.1 Development of an ILA modeling language

We developed *ILAng* [27], an ILA modeling framework to provide a formal logical foundation for ILA models, with applications in specification and verification. ILAng supports a formal intermediate representation for ILA models and provides its operational semantics. On the front-end, it includes converters from C/C++ and SystemC ILA-style models. At the back end, ILAng supports standard verification tools and their formats, e.g., Z3 [17], open-source model checking tools developed by the Stanford group [37], as well as commercial model checking tools [11].

A basic ILA model, like an ISA, includes the following: (i) a set of instructions; (ii) meaningful architectural state, i.e., state that is persistent between instructions at the architecture level, while abstracting away implementation state; and (iii) state update functions for each instruction. In addition to a basic block-level model, ILAng also supports hierarchy and concurrency to model complex designs with multiple ILA models. These features are described in detail in a journal article [28], which won the ACM Transactions on Design Automation of Electronic Systems

2020 Best Paper Award.

1.2.2 ILA-based Specification

Availability of high-level specifications is essential for an open-source hardware ecosystem to flourish. Our ILA-based high-level specifications can be used by system designers for system verification/validation, for HW/SW co-verification, for component replacement, and for incremental verification.

ILAs for Accelerators and Processors. We developed ILA-based specifications for a diverse set of publicly available hardware designs: application-specific accelerators for image processing (Gaussian Blur), machine learning (Restricted Boltzmann Machine), cryptography (AES, SHA), and a range of Rocket processor core designs (rocket, piccolo, nibbler) based on the RISC-V ISA. These specification models are publicly available [29], along with the formal semantics of the models [28].

ILA+MCM for specifying shared-memory interactions. Once the formal ILA model had been established, a natural next step was to formalize the shared-memory interactions between accelerators and programmable processors in Systems-on-a-Chip. This led to combining the operational ILA model with axiomatic memory-consistency models (MCM) to specify and verify these shared interactions [61].

ILAs for general hardware modules. We realized that while the ILA model had been developed for accelerators as an extension of the ISA, this could be extended to *general hardware modules* by treating the commands at their interface as instructions. This led to the use of the ILA for specifying and verifying general hardware modules, including an Ethernet MAC controller (designed by LeWiz, POSH performer) and OpenPiton L1.5 cache controller (designed by Wentzlaff group, POSH performer). The details are described in a paper [54], which won a Best Paper Award.

Automatic extraction of ILA models. To reduce the manual burden in specifying hardware designs, we developed techniques to automatically extract ILA models from legacy RTL. This work was done in two parts. The first was to automatically determine the set of architectural state variables (the variables persistent across instructions) [57], and the second was to extract the per-instruction update function for these architecture state variables [56].

1.2.3 ILA-based Verification

The generalization of ISAs to ILAs enabled us to use classic *refinement checking* techniques, originally developed for processor verification [10, 39], for accelerator and general module verification. As part of this, we extended the concept of a *refinement map* [60], which specifies *what* to check (i.e., the correspondence between output signals in the specification and the implementation) and *when* to check (i.e., when the instruction completes in terms of number of clock cycles or a predicate, etc.).

ILA-based functional verification. We used ILA-based refinement checking to prove correctness of several open-source designs for accelerators and processors [28], and other modules [54]. Given a refinement map between an ILA specification and an RTL/ILA implementation, the ILAng platform automatically generates properties to check the correctness of each instruction, thereby providing a method for complete functional verification.

Automatic generation of invariants. An important part of refinement checking for processor and accelerator implementation verification is the generation of appropriate invariants to constrain an arbitrary starting state in the refinement checking of each instruction. We did this through syntax-guided techniques leveraged from program synthesis [62]. We used our syntax-guided techniques to further improve a classic hardware model checking algorithm [59].
Compositional verification. We extended the ILA specification to model inter-ILA communication in terms of a generic valid/ready signal interface. Further, we used this interface specification to generate a set of interface checking properties that check that the communication between the RTL components is correct. This provides the following guarantee: if each RTL component is a refinement of its ILA specification and the interface checks pass, then the RTL composition is a refinement of the ILA composition. We applied the proposed methodology to six case studies including parts of large-scale designs such as parts of the FlexASR and NVDLA machine learning accelerators [55].
ILA-based Tandem simulation. An interesting application of the ILA methodology is in automating tan- dem simulation of processor/accelerator instruction-level models with RTL models. In tandem simulation, both models proceed in lockstep, one instruction at a time, and this is useful in design debugging. The ILA specification and refinement map were leveraged for complete automation of tandem simulation [52].
ILA-based GPU verification. The instruction-level abstraction offered by the ILA model was shown to be useful for modeling virtual instruction sets such as the NVidia PTX GPU virtual instruction set. This modeling was then used to verify concurrent PTX programs [53].

1.2.4 List of ILA-Related Tools and Publications in the POSH Project ILAng: ILA Modeling and Verification Platform

- Open source (MIT license): https://github.com/PrincetonUniversity/ILAng
- Docker: https://hub.docker.com/r/byhuang/ilang
- ILA modeling database (IMDB): https://github.com/PrincetonUniversity/IMDb
- Document: https://bo-yuan-huang.gitbook.io/ilang/
- API reference: https://princetonuniversity.github.io/ILAng-Doc/namespaceilang.html

1.3 List of Peer-Reviewed Publications

- Bo-Yuan Huang, Hongce Zhang, Pramod Subramanyan, Yakir Vizel, Aarti Gupta, Sharad Malik: Instruction-Level Abstraction (ILA): A Uniform Specification for System-on-Chip (SoC) Verification. ACM Transactions on Design Automation of Electronic Systems 24(1): 10:1-10:24 (2019). *Winner of the ACM TODAES 2020 Best Paper Award.*
- Hongce Zhang, Caroline Trippel, Yatin Manerkar, Aarti Gupta, Margaret Martonosi and Sharad Ma- lik: Integrating Memory Consistency Models with Instruction-Level Abstraction for Heterogeneous System-on-Chip Verification. In Proceedings of the International Conference on Formal Methods in Computer Aided Design (FMCAD), October 2018.
- Yue Xing, Bo-Yuan Huang, Aarti Gupta, Sharad Malik: A formal instruction-level GPU model

for scalable verification. In Proceedings of the International Conference on Computer-Aided Design (ICCAD) 2018: 130:1-130:8.

- Bo-Yuan Huang, Hongce Zhang, Aarti Gupta, Sharad Malik: ILAng: A Modeling and Verification Platform for SoCs Using Instruction-Level Abstractions. In Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS (1) 2019: 351-357.
- Hongce Zhang, Weikun Yang, Grigory Fedyukovich, Aarti Gupta, Sharad Malik: Synthesizing Environment Invariants for Modular Hardware Verification. In Proceedings of the International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI), 2020.
- Hongce Zhang, Aarti Gupta, Sharad Malik: Syntax-Guided Synthesis for Lemma Generation in Hardware Model Checking. In Proceedings of the International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI) 2021: 325-349.
- Yue Xing, Huaixi Lu, Aarti Gupta, Sharad Malik: Leveraging Processor Modeling and Verification for General Hardware Modules. In Proceedings of the International Conference on Design, Automation, and Test in Europe (DATE) 2021. *Winner of the 2021 Best Paper Award, in Track D: Design Methods and Tools.*
- Yu Zeng, Bo-Yuan Huang, Hongce Zhang, Aarti Gupta, Sharad Malik: Generating Architecture-Level Abstractions from RTL Designs for Processors and Accelerators Part I: Determining Architectural State Variables. In Proceedings of the IEEE/ACM International Conference on Computer Aided Design (ICCAD) 2021: 1-9.
- Yue Xing, Aarti Gupta, Sharad Malik: Generalizing Tandem Simulation: Connecting High-level and RTL Simulation Models. In Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC) 2022: 154-159.
- Yu Zeng, Aarti Gupta, Sharad Malik: Automatic Generation of Architecture-Level Models from RTL Designs for Processors and Accelerators. In Proceedings of the International Conference on Design, Automation, and Test in Europe (DATE) 2022: 460-465.
- Yue Xing, Huaixi Lu, Aarti Gupta, Sharad Malik: Compositional Verification Using a Formal Com- ponent and Interface Specification. IEEE/ACM International Conference on Computer Aided De- sign (ICCAD), October 2022.
- Bo-Yuan Huang, Hongce Zhang, Aarti Gupta, Sharad Malik: Generalizing the ISA to the ILA: A Software/Hardware Interface for Accelerator-rich Platforms. *Invited paper* at a special session in the ACM/IEEE Design Automation Conference (DAC), July 2023.

1.4 Analog Design and Modeling

For many reasons, including the increasing performance requirements and low cost of digital hardware, an increasing number of analog building blocks have rich connections with digital logic that "controls" their operation. These blocks depend on sophisticated calibration/adaptation algorithms for proper operation. This coupling of the analog and digital sections of an SoC makes it necessary to validate their combined operation, which is challenging because analog and digital circuits fail for different reasons, and thus traditionally use different tools and testing approaches for validation. Our POSH research effort was to address many of these challenges.

For digital systems, validation is all about test coverage. This need to explore the entire state space arises because the output result surface of a digital circuit isn't smooth; an incremental

change in an input could completely change the output, so one must look at **all** possible inputs to ensure correct behavior. Thus, advanced testing strategies employ formal methods like those explored in other parts of this research effort to guide the exploration or try to find specific inputs that violate assertions, and generally require complex validation frameworks.

Interestingly, analog circuits are nearly the opposite. Their analog nature implies that their output surface is smooth, since that is what makes them analog.[2] In fact, for most analog circuits, that surface is not only smooth but also nearly linear with respect to primary analog inputs. The smooth result surface means that the measured output of one set of inputs provides a lot of data about what the output will be for a different set of inputs. This means that generating a set of test vectors to cover a traditional analog circuit is not difficult. Instead, the complexity in analog testing comes from the non-discrete nature of the outputs. One needs to consider many more circuit inputs (supply voltage/noise, temperature, process, bias signals, etc.) and extract their effect on the circuit. As a result, analog circuit validation has focused more on creating the right high-fidelity model of the circuit and characterizing different "noise" sources.

To address this validation challenge, engineering teams have settled on creating "real number" functional models as a necessary piece of modern SoC verification, but the creation of functional models is still a challenging and specialized process. Knowledge of analog circuit behavior and digital simulation techniques are both necessary to create a good model, making it relatively difficult to find engineers suited for the task. As part of our research, we reviewed these SoTA approaches to mixed signal modeling in [51]. Our modeling effort took an existing approach to automated functional modeling, the DaVE ecosystem, which uses a template-based modeling effect, and made significant changes and improvements resulting in the tool Fixture [50]. Fixture creates a much more powerful template extension capability, allowing each template to cover a wider class of circuits. Fixture also generates a complete testbench for both the functional model and the SPICE netlist, making it easy to extract the functional model parameters and compare functional model fidelity.

While fixture provides high performance functional models, many top-level simulations are so com- plex that they need to be emulated. So in addition to our work on functional models, we also worked to create efficient mixed signal functional models for emulators. Through this research effort we have created what we believe is the first publicly available, complete framework for AMS emulation. The free, open-source framework consists of *msdsl* [24], a Python tool for generating synthesizable AMS models, and *anasymod* [44], a simulator-like abstraction of FPGA boards. A standalone synthesizable fixed- and floating-point library, *svreal* [25], is used in the model generation process. The flow of creating models for FPGA emulation [26] is depicted in Fig. 2. *msdsl* provides a set of functions that allows users to describe AMS blocks as *differential equations*, *netlists*, *transfer functions*, or *switched systems*. The generated synthesizable HDL then leverages *svreal* to switch between fixed-point and floating-point representation to allow the flexible trade-off between accuracy and emulation throughput.

[2]As Kim showed in [31] this smooth response curve may not be apparent when looking at the voltage (or current) vs. time waveforms. For these analog circuits, one first needs to transform the domain of the input and/or output of the system. For example, a VCO with a digital output doesn't seem linear, but if one looks at the phase of the generated output, it is the integral of a

(piecewise) linear function of the input voltage.

anasymod provides emulation infrastructure that manages the emulation timestep, emulation clock speed, and test interfaces; it also communicates with EDA tools to generate the FPGA emulation bitstream for a given FPGA board. Details and examples of how to use the emulation framework are described in [26].

This framework supports event-driven emulation with a spline-based analog waveform representation, as described in Herbst's thesis [23]. It has been successfully applied to six commercial designs, including an automotive magnetic sensor [43], and one academic design, a high-speed link receiver [31] which we developed as part of this research effort.

The final part of our effort in analog validation was to use these tools to build a high-speed serial link. Generally, design and validation of a high-speed serial link require lots of engineering effort, since its performance tightly depends on sophisticated analog and mixed-signal (AMS) circuit blocks. While digital designs are synthesized into standard gates and then laid out using an automatic placement and routing (PnR) tool, AMS blocks rely on a transistor-level custom design flow, which requires a significant amount of non-recurring engineering (NRE) cost.

Our research created a synthesizable and portable architecture for the AMS building blocks of an ADC-based serial link along with an automated design platform. The proposed architecture enables the link to be entirely described by a hardware description language (HDL), using a minimal amount of analog precision. This platform is called the Open-Source PHY. [32]. The proposed receiver architecture does not contain front-end amplifiers (CTLE, VGA) because they are tricky to implement in a synthesizable fashion. Moreover, it does not rely on DFE because it can be problematic for timing closure. Instead, the proposed architecture adopts a novel MLSD-based error correction technique to compensate for the performance degradation due to the absence of amplifiers and DFE. This link, called DragonPhy, was made from std cells, except for two cells which used a modified std cell design.

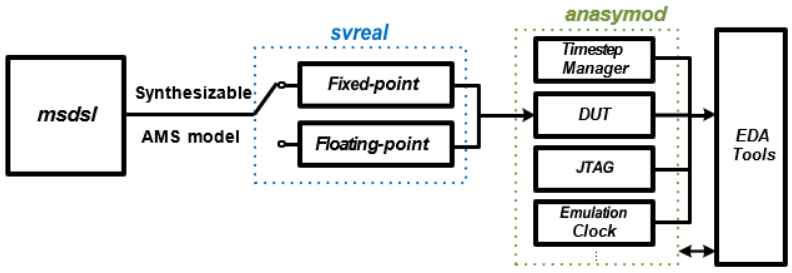

Figure 3: Framework of FPGA AMS Emulation Model Generation [26]

1.5 QED Techniques

While simulation-based verification has well-known issues of thoroughness, even today traditional formal verification is crippled by the following big challenges:

Challenge 1. Manually crafting extensive design-specific properties or full abstract functional specifications can be time-consuming and error-prone.

Challenge 2. Scaling to large designs (millions of gates) is difficult or infeasible using off-the-shelf formal tools.

We address these two problems with our suite of pre-silicon QED verification techniques. Symbolic Quick Error Detection (Symbolic QED) provides a thorough and fully automatable verification frame- work [34]. Unlike conventional verification, Symbolic QED does not require the formal specification of the processor implementation or design-specific properties. Given the Instruction-Set-Architecture, Symbolic QED can provide a thorough verification framework without any false fails. We have an automatic push-button Symbolic QED framework generator for any processor adhering to the RISC-V ISA.

An industrial case study with Infineon Technologies showed that Symbolic QED can catch all previously detected bugs and 7% more bugs in automotive grade micro-controllers in weeks vs. years of industrial grade verification [47]. **Currently, Symbolic QED verification has been adopted in their IoT pipeline and is providing** 10**× better productivity benefits.** Symbolic QED is provably sound and complete, under modest assumptions. Unlike traditional verification, Symbolic QED does not rely on the implementation details of the processor.

We have also mathematically investigated the thoroughness of Symbolic QED. Our work [35] shows how to guarantee soundness of Symbolic QED, which means that it will never create false fails. With the augmentation of a special state reset instruction, a simple-to-implement design for verification in any processor, Symbolic QED can be proven to be complete which means all logic bugs would be caught within the capabilities of the formal tool.

To address scalability issues of formal verification for processors, symbolic QED was improved to handle designs as large as 0.5 billion transistor SoC OpenSPARC T2 which would never load into the formal tool. Thus, end-to-end formal verification of such designs is practically impossible. With parital instantiation of the design, Symbolic QED could catch bugs in such a large design [48].

Another aspect of scalability is detecting bugs that are very deep (Trillion of inputs to trigger). Such

bugs are almost impossible for any verification to catch starting from reset. With symbolic starting states, the formal tool can choose a state very close to the bug and trigger the bug in a few clock cycles. However, this comes with the price of creating a lot of design and property specific constraints in traditional verification which makes it a cumbersome and error-prone task. With Symbolic QED a one-shot symbolic starting state verification has been demonstrated that does not require an iterative constraint generation [19]. This variant of Symbolic QED could catch 100% of the 350 Hardware Trojans (over Trillion clock cycles to trigger) in different processor designs.

To address the verification challenges for hardware accelerators (HAs), Accelerator Quick Error Detection (A-QED) was proposed [49]. It provides a provably sound and complete framework for non- interfering HAs, i.e., the output of the HA only depends on the current input irrespective of the context of other inputs. A-QED checks for self-consistency, more specifically, functional consistency, which checks that the same input in a sequence of inputs produces the same output.

Like Symbolic QED, it does not require understanding the intricate design implementation details. In a case study with the Stanford AHA group's memory controller unit, A-QED caught 13% more bugs over the previously detected ones in 1 day vs. a month of traditional verification (30-fold productivity benefits). Thus, A-QED addresses the challenge of manually crafting design-specific properties or full functional specifications.

To address the challenge of scalability, we developed a variant of A-QED called A-QED with De- composition (A-QED^2) [14]. It provides a systematic way of decomposing the HA into sub-accelerators which are then verified with A-QED. A-QED^2 has the same completeness guarantees as A-QED, which means that if A-QED can catch a bug in the full design, the bug would be caught by A-QED in one of the decompositions. Results on over 100 (buggy) versions of a wide variety of HAs with millions of logic gates demonstrate the effectiveness and practicality of A-QED^2.

2 REFERENCES

[1] M. Abadi and L. Lamport. The existence of refinement mappings. In *Proceedings of LICS*, pages 165–175, July 1988.

[2] C. W. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli. Satisfiability modulo theories. In *Handbook of Satisfiability*, pages 825–885. 2009.

[3] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In
Proceedings of TACAS, pages 193–207, 1999.

[4] N. Bjørner and A. Gurfinkel. Property Directed Polyhedral Abstraction. In *Proceedings of VMCAI*, pages 263–281, 2015.

[5] A. Bradley. SAT-based model checking without unrolling. In *Proceedings of VMCAI*, pages 70–87, 2011.

[6] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Trans. Computers*, (8):677–691, 1986.

[7] J. R. Burch and D. L. Dill. Automated verification of pipelined microprocessor control. In *Proceedings of the International Conference on Computer Aided Verification (CAV)*, pages 68–84, 1994.

[8] Cadence Design Systems, Inc. JasperGold: Formal Property Verification App., 2018. [Online]. Available:http://www.jasper-da.com/products/jaspergold-apps/, accessed on: 2022-04.

[9] R. Cavada, A. Cimatti, et al. The nuXmv symbolic model checker. In *Proceedings of CAV*, pages 334–342, 2014.

[10] A. Champion, A. Mebsout, C. Sticksel, and C. Tinelli. The Kind 2 model checker. In *Proceedings of CAV*, pages 510–517, 2016.

[11] S. Chattopadhyay, F. Lonsing, L. Piccolboni, D. Soni, P. Wei, X. Zhang, Y. Zhou, L. Carloni,
D. Chen, J. Cong, R. Karri, Z. Zhang, C. Trippel, C. Barrett, and S. Mitra. Scaling up hardware accelerator verification using a-qed with functional decomposition. In *2021 Formal Methods in Computer Aided Design (FMCAD)*, pages 42–52, 2021.

[12] A. Cimatti, A. Griggio, S. Mover, and S. Tonetta. Infinite-state invariant checking with IC3 and predicate abstraction. *FMSD*, (3):190–218, 2016.

[13] E. Clarke, T. Henzinger, et al. *Handbook of Model Checking*. 2018.

[14] L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In C. Ramakrishnan and J. Rehof, editors, *Tools and Algorithms for the Construction and Analysis*

of Systems, volume 4963 of *Lecture Notes in Computer Science*, chapter 24, pages 337–340. Springer Berlin / Heidelberg, Berlin, Heidelberg, 2008.

[15] N. Eén, A. Mishchenko, and R. K. Brayton. Efficient implementation of property directed reachability. In *Proceedings of FMCAD*, pages 125–134, 2011.

[16] K. Ganesan, F. Lonsing, S. S. Nuthakki, E. Singh, M. R. Fadiheh, W. Kunz, D. Stoffel, C. Barrett, and S. Mitra. Effective pre-silicon verification of processor cores by breaking the bounds of symbolic quick error detection, 2021.

[17] S. Ghilardi and S. Ranise. MCMT: A model checker modulo theories. In *Automated Reasoning*, pages 22–29, 2010.

[18] A. Goel and K. A. Sakallah. Model checking of Verilog RTL using IC3 with syntax-guided abstrac- tion. In *Proceedings of NFM*, pages 166–185, 2019.

[19] A. Goel and K. A. Sakallah. AVR: Abstractly Verifying Reachability. In *Proceedings of TACAS*, pages 413–422, 2020.

[20] S. Herbst. *An Open-Source Framework for FPGA Emulation of Analog/Mixed-Signal Integrated Circuit Designs*. PhD thesis, Stanford University, 2021.

[21] S. Herbst. msdsl, 2021.

[22] S. Herbst. svreal, 2021.

[23] S. Herbst, G. Rutsch, W. Ecker, and M. Horowitz. An Open-Source Framework for FPGA Emulation of Analog/Mixed-Signal Integrated Circuit Designs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Aug 2021.

[24] B.-Y. Huang, H. Zhang, A. Gupta, and S. Malik. ILAng: A modeling and verification platform for SoCs using instruction-level abstractions. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 351–357, 2019.

[25] B.-Y. Huang, H. Zhang, P. Subramanyan, Y. Vizel, A. Gupta, and S. Malik. Instruction-level abstraction (ILA): A uniform specification for system-on-chip (SoC) verification. *ACM Transactions on Design Automation of Electronic Systems*, 24(1), Dec. 2018.

[26] IMDb: ILA model database, n.d.

[27] D. Jovanovic and B. Dutertre. Property-directed k-induction. In *Proceedings of FMCAD*, pages 85–92, 2016.

[28] S. Kim. *Synthesizable Mixed-Signal Building Blocks for Open Source High Speed Serial Links*. PhD thesis, Stanford University, 2021.

[29] S. Kim, Z. Myers, S. Herbst, B. Lim, and M. Horowitz. Open-Source Synthesizable Analog Blocks for High-Speed Link Designs: 20-GS/s 5b ENOB Analog-to-Digital Converter and 5-GHz Phase Interpolator. In *2020 IEEE Symposium on VLSI Circuits*, pages 1–2, 2020.

[30] D. Kroening, A. Groce, and E. M. Clarke. Counterexample guided abstraction refinement via pro- gram execution. In *Proceedings of ICFEM*, pages 224–238, 2004.

[31] F. Lonsing, K. Ganesan, M. Mann, S. S. Nuthakki, E. Singh, M. Srouji, Y. Yang, S. Mitra, and C. Barrett. Unlocking the power of formal hardware verification with cosa and symbolic qed. In *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8. IEEE, 2019.

[32] F. Lonsing, S. Mitra, and C. Barrett. A theoretical framework for symbolic quick error detection. In *2020 Formal Methods in Computer Aided Design (FMCAD)*, pages 1–10, 2020.

[33] M. Mann, A. Irfan, et al. Counterexample-guided prophecy for model checking modulo the theory of arrays. In *Proceedings of TACAS*, pages 113–132, 2021.

[34] M. Mann, A. Irfan, F. Lonsing, Y. Yang, H. Zhang, K. Brown, A. Gupta, and C. Barrett. Pono: A flex- ible and extensible SMT-based model checker. In R. Leino and A. Silva, editors, *Proceedings of the 33rd International Conference on Computer Aided Verification (CAV '21), volume 12760 of Lecture Notes i −474. Springer International Publishing, July 2021.*

[35] M. Mann, A. Wilson, et al. Smt-Switch: A Solver-agnostic C++ API for SMT Solving. In Proceedings of SAT*, 2021.*

[36] P. Manolios and S. K. Srinivasan. A complete compositional reasoning framework for the efficient verification of pipelined machines. In Proceedings of the International Conference on Computer-Aided Design (ICCAD)*, pages 863–870, 2005.*

[37] K. McMillan. Symbolic model checking - an approach to the state explosion problem*. PhD thesis, Carnegie Mellon University, 1992.*

[38] K. L. McMillan. Interpolants and symbolic model checking. In Proceedings of VMCAI*, pages 89–90, 2007.*

[39] K. L. McMillan and O. Padon. Ivy: A multi-modal verification tool for distributed algorithms. In Proceedings of CAV*, pages 190–202, 2020.*

[40] G. Rutsch, S. Fontanesi, S. Herbst, S. Tan Hee Yeng, A. Possemato, G. Formato, M. Horowitz, and
W. Ecker. Boosting mixed-signal design productivity with FPGA-based methods throughout the chip design process. In Design and Verification Conference in Europe, 2020.

[41] G. Rutsch, S. Herbst, and S. Saravanan. anasymod, 2021.

[42] M. Sheeran, S. Singh, and G. Stålmarck. Checking safety properties using induction and a SAT- solver. In Proceedings of FMCAD, pages 108–125, 2000.

[43] J. P. M. Silva, I. Lynce, and S. Malik. Conflict-driven clause learning SAT solvers. In Handbook of Satisfiability, pages 131–153. 2009.

[44] E. Singh, K. Devarajegowda, S. Simon, R. Schnieder, K. Ganesan, M. Fadiheh, D. Stoffel, W. Kunz,
C. Barrett, W. Ecker, and S. Mitra. Symbolic qed pre-silicon verification for automotive micro- controller cores: Industrial case study. In 2019 Design, Automation Test in Europe Conference Exhibition (DATE), pages 1000–1005, 2019.

[45] E. Singh, D. Lin, C. Barrett, and S. Mitra. Logic bug detection and localization using symbolic quick error detection. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2018.

[46] E. Singh, F. Lonsing, S. Chattopadhyay, M. Strange, P. Wei, X. Zhang, Y. Zhou, D. Chen, J. Cong,
P. Raina, Z. Zhang, C. Barrett, and S. Mitra. A-qed verification of hardware accelerators. In 2020 57th ACM/IEEE Design Automation Conference (DAC), pages 1–6, 2020.

[47] D. Stanley. fixture: A library of templates for analog blocks and strategies to model them in a digital environment, 2021.

[48] D. Stanley, C. Wang, S.-J. Kim, S. Herbst, J. Kim, and M. Horowitz. Fast validation of mixed signal socs. IEEE Open Journal of the Solid-State Circuits Society, 1:184–195, 2021.

[49] Y. Xing, A. Gupta, and S. Malik. Generalizing tandem simulation: Connecting high-level and RTL simulation models. In Proceedings of the 27th Asia and South Pacific Design Automation Conference (ASP-DAC), pages 154–159, 2022.

[50] Y. Xing, B. Huang, A. Gupta, and S. Malik. A formal instruction-level GPU model for scalable verification. In Proceedings of the International Conference on Computer-Aided Design (ICCAD), page 130. ACM, 2018.

[51] Y. Xing, H. Lu, A. Gupta, and S. Malik. Leveraging processor modeling and verification for general hardware modules. In Design, Automation & Test in Europe Conference & Exhibition, (DATE)*, pages 1130–1135. IEEE, 2021.*

[52] Y. Xing, H. Lu, A. Gupta, and S. Malik. Compositional verification using a formal component and interface specification. In Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 72:1–72:9. ACM, 2022.*

[53] Y. Zeng, A. Gupta, and S. Malik. Automatic generation of architecture-level models from RTL designs for processors and accelerators. In Design, Automation & Test in Europe Conference & Exhibition, DATE*, pages 460–465. IEEE, 2022.*

[54] Y. Zeng, B. Huang, H. Zhang, A. Gupta, and S. Malik. Generating architecture-level abstractions from RTL designs for processors and accelerators part I: determining architectural state variables. In Proceedings of the IEEE/ACM International Conference On Computer-Aided Design, (ICCAD)*. IEEE, 2021.*

[55] H. Zhang, A. Gupta, and S. Malik. Syntax-guided synthesis for lemma generation in hardware model checking. In Proceedings of VMCAI*, 2021.*

[56] H. Zhang, A. Gupta, and S. Malik. Syntax-guided synthesis for lemma generation in hardware model checking. In Verification, Model Checking, and Abstract Interpretation - 22nd International Conference (VMCAI), Proceedings*, volume 12597 of* Lecture Notes in Computer Science*, pages 325–349. Springer, 2021.*

[57] H. Zhang and B.-Y. Huang. Refinement relation specification in ILAng, 2021.

[58] H. Zhang, C. Trippel, Y. A. Manerkar, A. Gupta, M. Martonosi, and S. Malik. ILA-MCM: Integrating memory consistency models with instruction-level abstractions for heterogeneous system-on-chip verification. In Proceedings of the International Conference on Formal Methods in Computer Aided Design (FMCAD)*, 2018.*

[59] H. Zhang, W. Yang, G. Fedyukovich, A. Gupta, and S. Malik. Synthesizing environment invariants for modular hardware verification. In Verification, Model Checking, and Abstract Interpretation - 21st International Conference (VMCAI), Proceedings*, volume 11990 of* Lecture Notes in Computer Science*, pages 202–225. Springer, 2020.*

LIST OF ABBREVIATIONS, ACRONYMS, AND SYMBOLS

ACRONYM	DESCRIPTION
AMS	Analog/Mixed-Signal
BDD	Binary Decision Diagrams
CEGAR	Counterexample-Guided Abstraction Refinement
DARPA	Defense Advanced Research Projects Agency
HWMCC	Hardware Model Checking Competition
ILAs	Instruction- Level Abstractions
SMT	Satisfiability Modulo Theories
SoC	Systems-on-Chip
UF	Uninterpreted Functions