V. International Conference on Electrical, Computer and Energy Technologies (ICECET 2025)
3-6 July 2025, Paris-France

"Test, Build, Deploy" - A CI/CD Framework for Open-Source Hardware Designs

Calvin Deutschbein
School of Computing and Information Sciences
Willamette University
Oregon, United States of America
0000-0003-1354-7200

Aristotle Stassinopoulos
School of Computing and Information Sciences
Willamette University
Oregon, United States of America
0009-0008-9913-3140

***Abstract*—Addressing TedX, Amber Huffman [20] made an impassioned case that "none of us is as smart as all of us" and that open-source hardware is the future. A major contribution to software quality, open source and otherwise, on the software side, is the systems design methodology of Continuous Integration and Delivery (CI/CD), which we propose to systematically bring to hardware designs and their specifications. To do so, we automatically generate specifications using specification mining, "a machine learning approach to discovering formal specifications" [1] which dramatically impacted the ability of software engineers to achieve quality, verification, and security. Yet applying the same techniques to hardware is non-trivial. We present a technique for generalized, continuous integration (CI) of hardware specification designs that continually deploys (CD) a hardware specification. As a proof-of-concept, we demonstrate Myrtha, a cloud-based, specification generator based on established hardware and software quality tools.**

***Index Terms*—Hardware, Security, Machine learning, Cloud computing, RISC-V, Open source, Containers, CI/CD, RTL, IaC, Specification Mining, Formal Verification.**

I. Introduction

Continuous Integration (CI) was first proposed in 1991 by Grady Booch for the software domain [4] as a once-per-day, automated, integration test. Since then, CI has exploded in popularity, especially as the broader "CI/CD" (for continuous integration and delivery) framework, a dominant framework for software quality assurance in recent years. In 2018, the launch of cloud-based solution "GitHub Actions" [27], by GitHub, the host of the Linux Kernel, Python, and tensorflow, led to a surge of CI/CD. But what about hardware?

Modern hardware designs are exceedingly complex [28], on the order of 10 billion transistors for consumer CPUs (central processing unit). To combat complexity, Hardware Description Languages (HDLs) enable hardware designers to design software by writing code. Many established language-based software tools may be adapted to HDL. Hardware trends have followed e.g. open source trends [20] and formal verification trends including specification mining [1].

This work was supported by NSF Award 171858

Many RISC-V [2] CPU designs are maintained, like software, under version control on GitHub, increasingly under automated testing. But we are aware of no continuous deployment framework for hardware, which is unsurprising for actually existing physical devices. Yet the product of hardware design is not only a physical device, but also a specification document that can be directly interpreted by the clients of hardware designs, such as compiler designers, embedded systems engineers, or security researchers.

In this work, we will demonstrate how to systematically apply CI, CD, and specification mining to hardware designs. We organize this around the inversion of the "Build, Test, Deploy" framework for CI/CD pipelines. For hardware, as we are deploying a specification generated through a testing process, we invert the first terms to "Test, Build, Deploy", and use specification mining as the build process, with standard CI and CD technologies. We perform all steps containerized on the cloud, for scalability and transparency. We recognize a simulation-only approach is insufficient for some hardware goals, but still supports of hardware quality assurance.

1) **Test**: Using established hardware tools and a testbench, we simulate a hardware design to generate a trace of execution as part of CI.
2) **Build**: Using specification mining, build a design specification from the trace data.
3) **Deploy**: Using GitHub Actions for CD, deliver the specification as a build artifact.

II. Methodology

We organize our methodology around managing primary (hardware design) and secondary (software) inputs and encapsulating to manage complexity.

A. Hardware Requirements

1) A Design: One or more HDL files.

The primary input is a hardware design specified in an HDL such as Verilog or VHDL. In general, we expect a design specified at register transfer level (RTL).

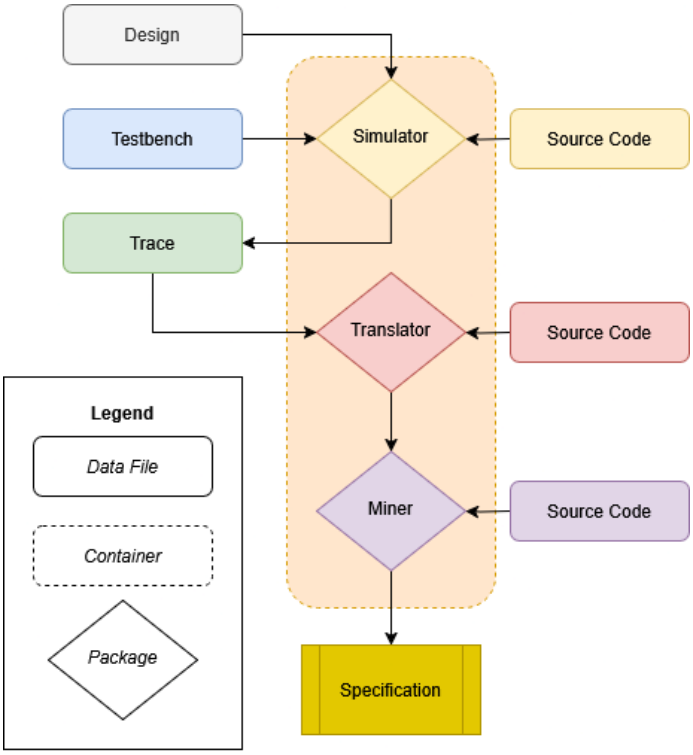

Fig. 1. A graph representation of the pipeline

2) A Testbench: One or more HDL files.

An HDL description of hardware cannot be executed and therefore cannot generate a trace of execution, which is necessary for specification mining. So, we introduce the additional requirement of a testbench. Testbench generation is a separate, active area of research [35] but we only require that some imperatives be dispatch to a hardware design from a simulation framework that may logged the hardware state.

In general, we find that testbenches are often maintained under version control in the same HDL as the design to which they accompany, as development without testbenches is exceedingly difficult and uncommon. For all designs we explored, we found testbenches easily.

3) A Simulator: Source code for some software which simulates hardware.

To generate a trace of execution, we must execute hardware in simulation (or generate an equivalent design with hardware monitors, a separate, active area of research [30]). For a cloud-based and scalable solution, we instead use simulation, which has limitations with respect to hardware designs but is suitable for generation of specifications.

In our experience, we found the best approach to simulation was via encapsulation, specifically containerization. We built our from source within a container and then removed the source code to reduce memory footprint. This led to lightweight, powerful containers with no external dependencies that could be easily deployed to cloud services, and completely abstracted the complexity of hardware simulation from our workflow.

B. Specification Mining Requirements

1) A Trace: An intermediate data file of unspecified type.

The hardware "test" phase terminates with the generation of a hardware trace of execution. In practice, these are often "value change dump" or .vcd files, which specify all changes to the internal state of a hardware design while executing some series of imperatives.

2) A Translator: An custom executable or script.

To our knowledge, there is no widely-used, general-purpose specification miner that accepts traces of execution from software designs, so we implemented our own translation framework, It transposes traces of hardware execution into a format consistent with traces produced by software for software specification miners. In practice, we translated from .vcd to a trace format for C/C++ executables, which was similar to HDLs in terms of data types.

3) A Miner: Source code for software which implements the specification mining machine learning process.

To generate a specification from a trace of execution, specification miners infer some universe of candidate properties and then systematically falsify candidate properties while traversing a trace(s) of execution. This process is parallelizable on many axes and may scale quite well, even on larger designs. While arbitrarily sophisticated machine learning techniques may be employed, the specification miners we surveyed tended to rely on heuristic-based algorithms, like k-means or hierarchical clustering.

As with the simulator, miners are often large and sophisticated pieces of software with complex build processes. In our case, we converged on a specification miner with a Java Virtual Machine (JVM) dependency, but otherwise maintained only a single Java ARchive or .jar file within the container.

C. The Pipeline

We present the pipeline visually in Fig. 1. Consistent with the practice of "Infrastructure as Code (IaC)", all software components implementing the pipeline into a single container image which contains a hardware simulator, a translator, and a specification miner. Our also contains a JVM and a Python installation, which we leveraged for our translator, but any pipeline stages could be implemented as binaries, as was the hardware simulator.

This container image then becomes a single, separately maintained dependency for both the hardware "test" stage which produces a trace and the software "build" stage which produces a specification. We then simply provide the remaining inputs - the hardware design and the testbench - to this container and execute brief script which generates and deploys the specification, in our case as a single .yml (variously "yet another markup language" or "yaml ain't markup language") file following GitHub Action standards in order to "deploy".

We had one remaining input not covered here, in that in all cases we additionally used a Makefile to generate traces. Many hardware designs already provided Makefiles which we adapted to make within our workflows.

III. Implementation

We implement our methodology with "Mythra", an open source package as a container image maintained publicly on GitHub, itself under CI/CD via GitHub actions to GitHub Container Registery (GHCR) [1].

A. Simulator: Icarus Verilog

Icarus Verilog is a free and open source Verilog compiler under the GPL license, maintained on GitHub [2]. "Icarus Verilog is not aimed at being a simulator in the traditional sense, but a compiler that generates code employed by back-end tools." and in our case is suitable for creating a .vcd file given some verilog input.

Both Verilog and VHDL are popular for hardware designs, but most design we were aware of used Verilog (or SystemVerilog). Our toolchain should be fully compatible with other compilers, like Verilator or commercial tools.

Both Verilator and Icarus Verilog use the .vcd text-based format, incurring write-speed limits at one eighth of the speed of a binary representation (when printing textual binary, one bit of information incurs eight bits of storage). Separately, there is no compelling reason to store the .vcd representation at all, versus streaming directly into a machine learning framework. We hope to approach data streaming in future research, and have already adapted our custom translator for streaming data.

B. Translator: Custom Python

We are aware of no specification miner for .vcd files, and prefer to use established miners as a proof-of-concept regardless. So we required an intermediate stage translating from .vcd to some format suitable for input into our miner. Our miner required ".decls" declaration files, enumerating variables, and ".dtrace" Daikon trace files, enumerating the values of all variables at every time point. We note that these files are also text-based formats, with the accompanying speed limitations.

We implemented this functionality with a Python script which streamed text data file-to-file maintaining an internal state only large enough to track a current value for each register. Our performance metrics prior to the use of streaming suggested a performance bottleneck in the translation stage due to memory footprint, and the streaming implementation we shift our bottleneck to Daikon, our externally maintained specification miner.

We recognize including a Python installation in a container to use a single script is an inefficient use of container size. We hope to refactor into an executable in future work. Separately, we may wish to preserve the Python installation and leverage Python APIs to GPU acceleration in the machine learning stage, in which case the primary change would be to use PyArrow or Torch data structures rather than Python built-ins.

C. Miner: The Daikon invariant detector

"Daikon [14] is an implementation of dynamic detection of likely invariants; that is, the Daikon invariant detector reports likely program invariants. An invariant is a property that holds at a certain point or points in a program; these are often seen in assert statements, documentation, and formal specifications... Daikon can detect properties in C, C++,... and in other data sources. (Dynamic invariant detection is a machine learning technique that can be applied to arbitrary data.) It is easy to extend Daikon to other applications." We agree.

We regard specification mining as a separate, ongoing area of research [21] and simply use Daikon in our pipeline given our understanding of its popularity. Daikon is implemented as .jar file. Its C/C++ language front-end "Kvasir" is well suited to model HDL registers of fixed-width binary values. We made only one specification-relevant design decision when translating .vcd files to Daikon traces, which was how to regard the "x" unknown and "z" high impedance values that exist at a hardware level but do not persist at software level. By treating binary data as unsigned values and we use the sign bit to indicate hardware-specific values. This results in minor information loss conflating "x" and "z", and may require wider words for some values, but was suitable for our purposes.

Consistent with the Daikon documentation, we do not build from source but use a packaged release.

In our current pipeline, this stage has by far the highest time cost, a Daikon reads the entire trace data in a text-based format before processing. Daikon usage is motivated by a desire for consistency with existing tools, but we note Daikon uses algorithms for k-means or hierarchical clustering that could proceed under GPU-accelerated binary data, possible streaming data, through PyTorch or CUDA C++, and any pipeline will likely have a C/C++ dependency for its hardware simulator. We regard this as an area of future work.

D. The Pipeline

We present the pipeline visually in Fig. 2. It is identical to the proposed methodology except for 3 changes:

1) A Makefile is used in .vcd generation.
2) The translator is a Python script, rather than an executable.
3) The miner as a JVM package, rather than an executable.

IV. Evaluation

We developed our implementation over PicoRV32 [3], "a CPU core that implements the RISC-V RV32IMC Instruction Set", for which we can report design size, lines of code (LoC), execution times, and specification output. Some evaluations are public via the HWCICD organization [4].

[1] https://github.com/hwcicd/myrtha/pkgs/container/myrtha
[2] https://github.com/steveicarus/iverilog
[3] https://github.com/YosysHQ/picorv32
[4] https://github.com/hwcicd

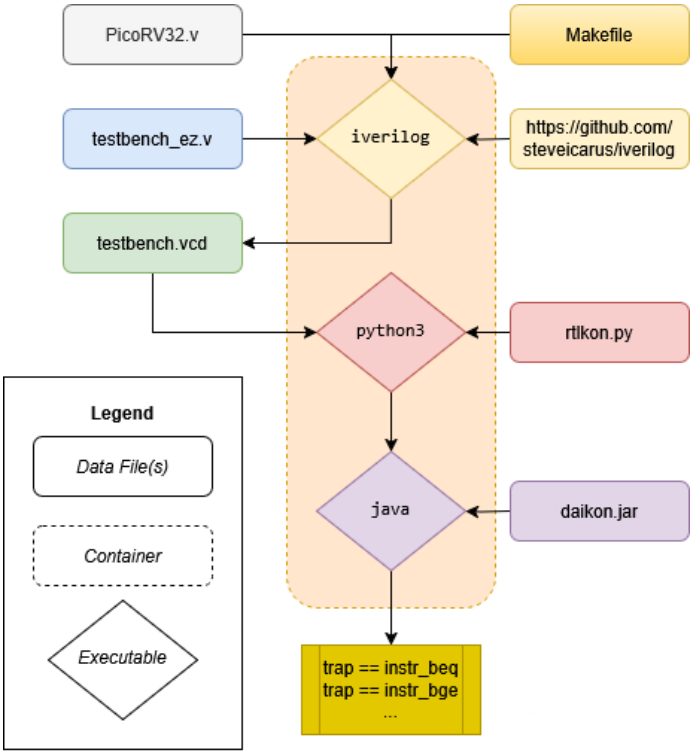

Fig. 2. A graph representation the Mythra implementation

A. PicoRV32

PicoRV32 contains 232 registers in 3049 lines of Verilog code. Its accompanying testbench is 86 lines of Verilog and runs for 2201 cycles. Our performance bottleneck, reading the .dtrace file, scales with the product of unique registers times clock cycles, and is most visible by observing the disk size of the .vcd, .decls, and .dtrace file. The .vcd also scales with this product, but non-linearly due to tracking on value changes, rather than clock cycles. The .decls scales only with design size. We present these size measures in Tab. I.

	lines	words	bytes
.vcd	30356	46200	269184
.decls	2219	4434	39059
.dtrace	2936134	2931732	17474090

TABLE I
PICORV32 TRACE DATA

B. Myrtha

Mythra is implemented as a package, structured over a total of 144 LoC across Python (71), a Containerfile (33), GitHub .yml workflow (32) , and Makefile (8). In general, it takes on the order of 5 minutes or 300 seconds to build the package. Over ten runs on GitHub, Myrtha built in on average in 345 seconds (349 median) with a standard deviation of 27 seconds. We built locally on a Linux device in 339 seconds. Using Docker instead of Podman, we built on Google Cloud Platform (GCP) in 511 seconds and on a local Windows device in 404 seconds. Build times were mostly dominated by compiling Icarus Verilog from source, which required both a length compilation process and downloading a number of packages. These times are summarized in Fig. 3.

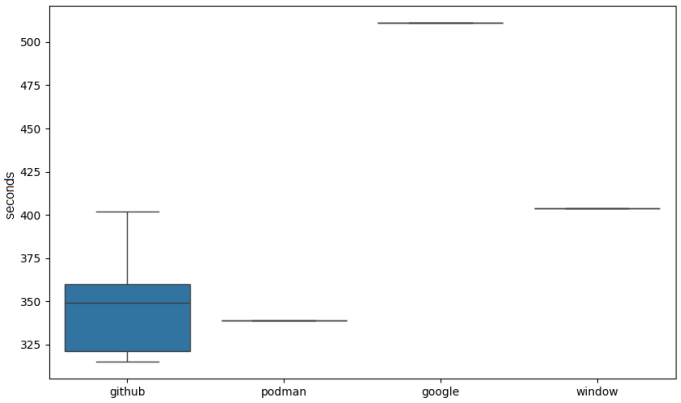

Fig. 3. Myrtha build times by platform.

Myrtha is a 1.72 GB image over an Ubuntu base. In future efforts, we hope to shift to an Alpine base and minimize usage of Python, the JVM, and elements of Ubuntu build-essentials not needed by hardware compilers to achieve a lower memory footprint. In practice, the Myrtha container size is reasonable for our application and did not constitute a performance bottleneck.

C. CI/CD

To perform CI/CD via Myrtha over PicoRV32, we forked the main PicoRV32 repository to our GitHub organization and update the Makefile and workflow. In total, we added 3 lines to the Makefile generating a specification and developed a 16 line "myrtha.yml" workflow. GitHub actions ran an average of 39.1 seconds (38 median) with a standard deviation of 3.3 seconds. We expect these times are dominated by initializatio, as the workflow runs on the default ubuntu:latest virtual machine hosting the Mythra image. Running locally without a VM, times always ranged from 4 to 4.5 seconds.

In future research, we hope to generalize our method to other CI/CD platforms, such as GitLab, which maintains both a cloud-hosted and a community edition, which may be hosted locally, so we can gain more detailed performance metrics over what is driving costs during the CI/CD hardware stages.

D. Specification

The output specification is a 4059 line text file summarizing binary equality and inequality operations between registers, modular relations between registers, and linear combinations of registers. This output is suitable to be regarded as a specification of the agreements maintained by PicoRV32, or as an input to a more mature specification generation, such as the security specification generators that derive security agreements from this form of output [10]–[12]. We present a few example invariants in Fig. 4.

```
4.294967283E9 * dbg_valid_insn +
  decoded_imm - 4.294967284E9 *
  is_lui_auipc_jal - 4.294967283E9 == 0
mem_wordsize % q_insn_opcode == 0
trap == eoi
```

Fig. 4. Example Specifications

E. Evaluation over Holdout Designs

After developing Myrtha alongside PicoRV32, we tested our approach over other desings. We applied the pipeline to AKER [5], a design and verification framework for SoC access control [26], and NERV, "a very simple single-stage RV32I processor.[6]". We use a closed-source testbench for AKER so many only report results.

1) AKER: We use a closed-source testbench for AKER and only report the results. With two changes, the entire pipeline ran without issue on the first attempt. To apply Myrtha to AKER, we used the exact Makefile and Myrtha.yml file used with PicoRV32, updating two lines of code, both within the Makefile:

1) **testbench.vcd target**: We updated the dependencies to refer to the AKER modules.
2) **testbench.vcd rule**: We added the -g2012 flag to iverilog to compile the .sv SystemVerilog testbench.

AKER contains 432 registers in two files totaling 2002 Verilog LoC. Its accompanying testbench is 527 lines of SystemVerilog and runs for 1055 cycles. We present these size measures in Tab. II. Vs. PicoRV32, as predicted, the .decls roughly doubled in size due to the doubling number of registers, but the .dtrace file remained roughly the same size as the clock cycles halved, preserving the product. With similar trace size, we save similar time cost in GitHub actions, with average 41.7 seconds (median 41) with a standard deviation of 3.1 seconds.

	lines	words	bytes
.vcd	6290	10990	53007
.decls	3519	7034	62829
.dtrace	2230270	2228160	13840288

TABLE II
AKER TRACE DATA

Local times always ranged from 6 to 6.5 seconds, a roughly one third time increase consistent with a .decls read bottleneck as .decls size increasing by roughly one third. The spec was 5580 lines, unsuprisingly larger given the doubling of registers but not scaling linearly with design size. Separately, longer traces tend to have fewer properties (as they are falsified over time) and the AKER testbench is shorter. Collectively, these metrics are a positive indicator for our scalability. Additionally, AKER required no changes to Myrtha, so there no marginal cost to pipeline management for placing this additional designs under CI/CD.

[5]https://github.com/KastnerRG/AKER-Access-Control
[6]https://github.com/YosysHQ/nerv

2) NERV: To apply Myrtha to NERV, we used the exact Makefile and Myrtha.yml file used with PicoRV32, updating two lines of code, both within the Makefile:

1) **testbench.vcd target**: We updated the dependencies to refer to the AKER modules.
2) **testbench.vcd rule**: We adapted the testbench rule from the NERV repository.

We encountered one bug with NERV, which used Verilog features shifted from warnings to errors by the latest release of Icarus Verilog. We rolled back to v11, an earlier stable branch, by changing one line of code in the Myrtha Containerfile and rebuilding. An earlier version of Myrtha had already used v11, but we switched to main branch to reduce LoC. Otherwise, the pipeline ran without issue and the new Myrtha container was suitable for PicoRV32 and AKER as well.

NERV contains 549 registers in SystemVerilog 1267 LoC. Its accompanying testbench is 155 SystemVerilog 1267 LoC and runs for 19 cycles. We present these size measures in Tab. III. GitHub actions run in average 36.8 seconds (median 36) with a standard deviation of 4 seconds.

	lines	words	bytes
.vcd	2248	6689	34848
.decls	5379	10754	101427
.dtrace	61370	61332	483308

TABLE III
NERV TRACE DATA

V. Related Work

Hardware Specification Mining: Early hardware specification leveraged known patterns, such as one-hot encoding [13], [17]. Other researchers applied data mining techniques [6], [18], [22], or temporal logics [7]–[9], [11]. Security researchers have manually identified properties [3], [5], [19]. automated generation for RISC [34] and CISC [12] designs and discovered subsets of hyperproperties [10], [24].

Specification Mining: Ammons et al. introduced specification mining [1] a launched a rich research direction across static and dynamic analysis [32], imperfect traces [33], and complex traces [15], [16], [25]. Perhaps the most widely known miner, Daikon [14] approached specification mining as invariant detection.

Hardware CI/CD Pipelines: Despite widespread adoption in software, to the best of our knowledge there is minimal research on continious integration for hardware design specification, outside of a few examples of Continious Integration/Continous Delivery (CI/CD) approaches to the embedded space [23], [31], which explores software and hardware together. By contrast, there is ample research on hardware acceleration for software CI/CD [29].

VI. Conclusion

We have proposed "Test, Build, Deploy" and demonstrate Myrtha, a proof-of-concept for a CI/CD cloud-based machine learning framework that generalized to other hardware designs.

ACKNOWLEDGEMENTS

We thank the NSF for funding. We thank Intel Corporation, the Kastner Research Group, and the HWSec@UNC research group for helpful discussions. We thank our undergraduate researchers, Kendall Leonard and Hannah Pahama, for suggesting the use of containerization and "ClickOps" to make hardware research more accessible.

REFERENCES

[1] Glenn Ammons, Rastislav Bodík, and James R. Larus. Mining specifications. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '02, page 4–16, New York, NY, USA, 2002. Association for Computing Machinery.

[2] K. Asanović and D. A. Patterson. Instruction sets should be free: the case for RISC-V. Technical Report UCB/EECS-2014-146, Department of Electrical Engineering and Computer Science, University of California, Berkeley, Berkeley, CA, USA, August 2014.

[3] M. Bilzor, T. Huffmire, C. Irvine, and T. Levin. Security checkers: Detecting processor malicious inclusions at runtime. In *International Symposium on Hardware-Oriented Security and Trust (HOST)*, pages 34–39. IEEE, June 2011.

[4] G. Booch. *Object Oriented Design with Applications*. Benjamin/Cummings series in Ada and software engineering. Benjamin/Cummings Publishing Company, 1991.

[5] Michael Brown. Cross-validation processor specifications. Master's thesis, University of North Carolina at Chapel Hill, 2017.

[6] Po-Hsien Chang and Li C Wang. Automatic assertion extraction via sequential data mining of simulation traces. In *Proceedings of the 15th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 607–612. IEEE, 2010.

[7] A. Danese, T. Ghasempouri, and G. Pravadelli. Automatic extraction of assertions from execution traces of behavioural models. In *Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 67–72, March 2015.

[8] A. Danese, G. Pravadelli, and I. Zandonà. Automatic generation of power state machines through dynamic mining of temporal assertions. In *Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 606–611, March 2016.

[9] A. Danese, N. D. Riva, and G. Pravadelli. A-TEAM: Automatic template-based assertion miner. In *Proceedings of the 54th Design Automation Conference (DAC)*, pages 1–6. ACM/EDAC/IEEE, June 2017.

[10] Calvin Deutschbein, Andres Meza, Francesco Restuccia, Ryan Kastner, and Cynthia Sturton. Isadora: Automated information flow property generation for hardware designs. In *Proceedings of the 5th Workshop on Attacks and Solutions in Hardware Security*, ASHES '21, page 5–15, New York, NY, USA, 2021. Association for Computing Machinery.

[11] Calvin Deutschbein and Cynthia Sturton. Mining security critical linear temporal logic specifications for processors. In *2018 19th International Workshop on Microprocessor and SOC Test and Verification (MTV)*, pages 18–23, 2018.

[12] Calvin Deutschbein and Cynthia Sturton. Evaluating security specification mining for a cisc architecture. In *2020 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 164–175, 2020.

[13] E. El Mandouh and A. G. Wassal. Automatic generation of hardware design properties from simulation traces. In *International Symposium on Circuits and Systems (ISCAS)*, pages 2317–2320. IEEE, 2012.

[14] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1-3):35–45, December 2007.

[15] Mark Gabel and Zhendong Su. Javert: Fully automatic mining of general temporal properties from dynamic traces. In *Proceedings of the 16th International Symposium on Foundations of Software Engineering (FSE)*, pages 339–349. ACM, 2008.

[16] Mark Gabel and Zhendong Su. Symbolic mining of temporal specifications. In *Proceedings of the 30th International Conference on Software Engineering (ICSE)*, pages 51–60. ACM, 2008.

[17] Sudheendra Hangal, Sridhar Narayanan, Naveen Chandra, and Sandeep Chakravorty. IODINE: A tool to automatically infer dynamic invariants for hardware designs. In *Proceedings of 42nd Design Automation Conference (DAC)*. IEEE, 2005.

[18] Samuel Hertz, David Sheridan, and Shobha Vasudevan. Mining hardware assertions with guidance from static analysis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 32(6):952–965, 2013.

[19] Matthew Hicks, Cynthia Sturton, Samuel T. King, and Jonathan M. Smith. SPECS: A lightweight runtime mechanism for protecting software from security-critical processor bugs. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 517–529. ACM, 2015.

[20] Amber Huffman. How open-source hardware is changing our future, tedxportland, Sep. 2024.

[21] Caroline Lemieux, Dennis Park, and Ivan Beschastnikh. General ltl specification mining (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 81–92. IEEE, 2015.

[22] Wenchao Li, Alessandro Forin, and Sanjit A Seshia. Scalable specification mining for verification and diagnosis. In *Proceedings of the 47th design automation conference*, pages 755–760, 2010.

[23] Shalu Prakasia. Investigating hardware simulation for continuous integration/continuous delivery, 2024.

[24] Mayank Rawat, Sujit Kumar Muduli, and Pramod Subramanyan. Mining hyperproperties from behavioral traces. In *2020 IFIP/IEEE 28th International Conference on Very Large Scale Integration (VLSI-SOC)*, pages 88–93, 2020.

[25] G. Reger, H. Barringer, and D. Rydeheard. A pattern-based approach to parametric specification mining. In *28th International Conference on Automated Software Engineering (ASE)*, pages 658–663. IEEE/ACM, 2013.

[26] Francesco Restuccia, Andres Meza, and Ryan Kastner. Aker: A design and verification framework for safe and secure soc access control. In *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pages 1–9, 2021.

[27] Sk Golam Saroar and Maleknaz Nayebi. Developers' perception of github actions: A survey analysis. In *Proceedings of the 27th International Conference on Evaluation and Assessment in Software Engineering*, EASE '23, page 121–130, New York, NY, USA, 2023. Association for Computing Machinery.

[28] Martin Schoeberl. Open-source hardware design.

[29] Mojtaba Shahin, Muhammad Ali Babar, and Liming Zhu. Continuous integration, delivery and deployment: A systematic review on approaches, tools, challenges and practices. *IEEE Access*, 5:3909–3943, 2017.

[30] Nikhilesh Singh, Shagnik Pal, Rainer Leupers, Farhad Merchant, and Chester Rebeiro. Promise: A programmable hardware monitor for secure execution in zero trust networks. *IEEE Embedded Systems Letters*, 16(4):433–436, 2024.

[31] Mayuri Talekar and Varsha K. Harpale. Ci-cd workflow for embedded system design. In *2023 7th International Conference On Computing, Communication, Control And Automation (ICCUBEA)*, pages 1–3, 2023.

[32] Westley Weimer and George C. Necula. Mining temporal specifications for error detection. In *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 461–476. Springer-Verlag, 2005.

[33] Jinlin Yang, David Evans, Deepali Bhardwaj, Thirumalesh Bhat, and Manuvir Das. Perracotta: Mining temporal API rules from imperfect traces. In *Proceedings of the 28th International Conference on Software Engineering (ICSE)*, pages 282–291. ACM, 2006.

[34] Rui Zhang, Natalie Stanley, Christopher Griggs, Andrew Chi, and Cynthia Sturton. Identifying security critical properties for the dynamic verification of a processor. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 541–554. ACM, 2017.

[35] Ziyue Zheng, Xiangchen Meng, and Yangdi Lyu. Ape-fv: Concolic testing for rtl functional verification using adaptive path exploration. In *2024 IEEE 42nd International Conference on Computer Design (ICCD)*, pages 373–380, 2024.