RTL Bug Localization Through LTL Specification Mining (WIP)

Vighnesh Iyer
vighnesh.iyer@berkeley.edu
University of California, Berkeley

Donggyu Kim
University of California, Berkeley
dgkim@eecs.berkeley.edu

Borivoje Nikolic
University of California, Berkeley
bora@eecs.berkeley.edu

Sanjit A. Seshia
University of California, Berkeley
sseshia@eecs.berkeley.edu

ABSTRACT

As the complexity of contemporary hardware designs continues to grow, functional verification demands more effort and resources in the design cycle than ever. As a result, manually debugging RTL designs is extremely challenging even with full signal traces after detecting errors in chip-level software simulation or FPGA emulation. Therefore, it is necessary to reduce the burden of verification by automating RTL debugging processes.

In this paper, we propose a novel approach for debugging with the use of LTL specification mining. In this approach, we extract fine-grained assertions that are implicitly encoded in the RTL design, representing the designer's assumptions, to localize bugs that are only detected when high-level properties are violated from long-running full-system simulations. We employ template-based RTL spec mining to infer both safety and bounded liveness properties. We propose strategies to convert multi-bit signals to atomic propositions based on common RTL design idioms such as ready-valid handshakes and specific state transitions using automatic static analysis.

Our initial results with a tiny RISC-V core design show that this methodology is promising for localizing bugs in time and space by demonstrating that the mined fine-grained LTL properties are violated before a high-level test failure condition occurs, such as a timeout or hanging, and can point to specific lines of suspect RTL.

CCS CONCEPTS

• **Hardware → Functional verification**; *Semi-formal verification*; *Assertion checking*.

KEYWORDS

specification mining, bug localization, RTL verification

ACM Reference Format:
Vighnesh Iyer, Donggyu Kim, Borivoje Nikolic, and Sanjit A. Seshia. 2019. RTL Bug Localization Through LTL Specification Mining (WIP). In *17th ACM-IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE '19), October 9–11, 2019, La Jolla, CA, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3359986.3361202

MEMOCODE '19, October 9–11, 2019, La Jolla, CA, USA

ACM ISBN 978-1-4503-6997-8/19/10…$15.00
https://doi.org/10.1145/3359986.3361202

1 MOTIVATION

It is no secret that functional verification is the main bottleneck of the modern SoC design process, requiring huge effort and resources in the design cycle [2]. As hardware designs are getting more sophisticated, RTL debugging is much more challenging. Even if we have full signal traces for errors that only manifest in full-system software simulation or FPGA emulation, it is mentally demanding to localize their root causes in the RTL design due to its complexity.

For example, Kim et al. [7] report assertion failures when the out-of-order RISC-V processor BOOM [1] was emulated on an FPGA, while executing the SPEC2006 benchmark suite. Some assertions were fired when the pipeline was hung, but such a failure condition has many potential causes. Even with a full waveform extracted from emulation, it was extremely challenging for the designer to localize the RTL bug since manual inspection of all possible causes in the complex hardware design requires significant time and effort.

While designer specified assertions are useful for catching errors, they are typically high-level and do not direct the designer to where a bug originated. While it is easy to check high-level properties at runtime with low overhead, checking violations of fine-grained properties caused by odd interactions in failing tests is also crucial for bug localization. In this paper, we propose the use of specification mining to derive fine-grained design properties that help localize a bug.

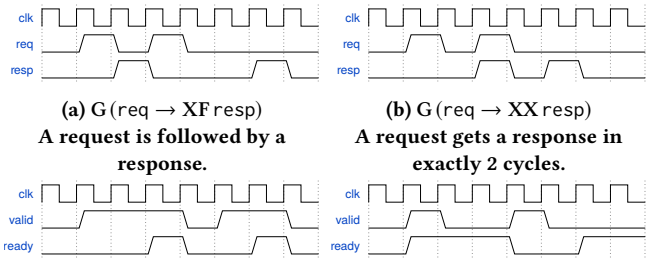
1.1 Hypothesis

We hypothesize that if a test fails on a mature design that passes many other tests, an *implicit assumption* the designer made about the design was likely violated. These assumptions are often not made explicit and usually describe common RTL transition patterns, such as the ones in Figure 1. Linear temporal logic (LTL) is a suitable formalization of these RTL idioms that can describe many meaningful properties.

We can extract these designer assumptions from waveforms of "normal" design behavior as *mined specifications*. These specifications can be added to the RTL design as assertions to catch anomalies earlier and *with greater locality*.

1.2 Prior Work

SAT-solver-based bug localization is a popular technique for hardware debugging. Smith et al. [11] present a technique to localize post-fabrication defects by solving a SAT problem given a set of



(a) G (req → XF resp)
A request is followed by a response.

(b) G (req → XX resp)
A request gets a response in exactly 2 cycles.

(c) G (valid → X (valid U ready))
Once valid is asserted, it remains high until ready is asserted.

(d) G ((valid ∧ ready) → (X ready ∨ XX ready))
After a R/V transaction, the slave is ready within 2 cycles.

Figure 1: Examples of common RTL idioms that can be represented as LTL properties.

failing test vectors. REDIR [4] uses a combination of mux instrumentation and SAT solving, along with executed statement analysis from RTL simulation to localize injected design bugs. Mux instrumentation involves adding bug models to the RTL design that could rectify common design bugs, such as using the wrong logical operator, and adding a mux to select between the bug model and the existing RTL which can be controlled by the SAT solver. Mirzaeian et al. [10] also use SAT-solver-based techniques with word-level mux instrumentation and a fault candidate list and demonstrated a technique to whittle down the candidates. While these techniques can enable *spatial* bug localization, they scale poorly for large sequential designs where many scenarios could have caused the bug and the design has to be unrolled for many cycles to reveal buggy behavior.

Specification mining has been used for analyzing behavior, constructing models, and evaluating test suites for software. Perracotta [14] mines properties from software execution traces and property templates, and can handle mining from traces where a property is occasionally violated. van der Aalst et al. [12] develop a general LTL mining algorithm on finite-length traces and mixes temporal operators with universal and existential quantification. Texada [8] introduces a spec mining tool that can mine any LTL formula with a powerful recursive algorithm, and introduces notions of confidence, support, and support potential which are particularly useful when evaluating properties over finite-length traces.

Specification mining has also been used for hardware design, often with the intent of constructing a formal specification of the system. IODINE [3] provides many spec mining analyses including extracting req-ack pairs, mutexes, one-hot signals, and scoreboards/FIFOs from analyzing waveforms. The tool was able to successfully extract FSM transitions and mine implicit low-level properties of several on-chip protocols, and revealed deficiencies in test coverage. Li et al. [9] introduce LTL templates and the notion of delta traces to handle multi-bit signals, applied them to mine specs on various RTL designs, and demonstrated fault localization using the mined properties. GoldMine [13] combines constrained random stimulus testing, static analysis, and spec mining with a formal verification engine to produce a set of verified properties for an RTL design.

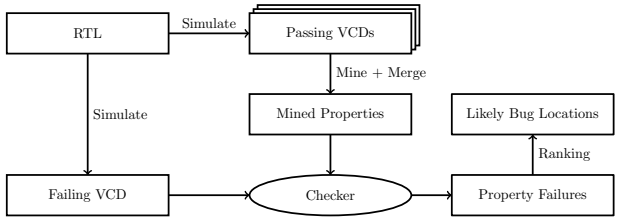


Figure 2: Bug localization flow using specification mining.

The key insight of our approach is that specification mining for bug localization is distinct from its use in deriving high-level designer-understandable formal specifications. Our goal is to extract "normal" design behavior, constrain it narrowly, and identify anomalies and deviations in error traces. Therefore, we aim to mine as many fine-grained properties as possible.

2 APPROACH

We begin with an RTL design and a set of waveforms from RTL simulations. Waveforms from passing tests are fed into a specification miner which produces a set of mined properties based on the observed "normal" behavior. The mined properties are checked on a waveform from a failing test to identify any anomalies and localize the RTL bug. The flow is summarized in Figure 2.

2.1 LTL Formula Templates

The mined properties are expressed as LTL formulas. We consider these LTL formula templates from [9] where a and b are *atomic propositions* (APs) that are a function of the signals in the RTL design.

- *Alternating* (**A**): $\neg a \mathbf{W} b \wedge \mathbf{G}((a \rightarrow \mathbf{X}(\neg x \mathbf{U} b)) \wedge (y \rightarrow \mathbf{X}(\neg a \mathbf{W} b)))$
- *Until* (**U**): $\mathbf{G}\,(a \rightarrow \mathbf{X}\,(a\,\mathbf{U}\,b))$
- *Next* (**N**): $\mathbf{G}\,(a \rightarrow \mathbf{X}\,b)$
- *Eventual* (**E**): $\mathbf{G}\,(a \rightarrow \mathbf{XF}\,b)$

The *Alternating* template captures a combination of a mutex property (a and b cannot be true simultaneously) and an ordering property (once a switches from true to false, b must transition from false to true to false before a can become true again).

These templates can capture a wide range of behavior when combined with various ways of constructing APs from signals in the design (Section 2.3).

LTL has two major caveats that are addressed in our approach:

- LTL formulas are defined over traces of infinite length, whereas simulation waveforms consist of finite-length traces.
- LTL formulas are defined over atomic propositions (boolean expressions), whereas an RTL design contains arbitrary-length bitvectors which must be transformed into meaningful atomic propositions.

2.2 LTL on Finite Traces

Typically, LTL formulas are evaluated on models by conversion to a Büchi automata and checking if its accepting states are visited infinitely often. This formulation is useful for model checking, but

is not appropriate for monitoring the formula on a finite-length trace.

On finite traces, safety properties can be falsified, but liveness properties are unfalsifiable. In our approach, the **F** operator is interpreted as a *bounded liveness* operator. When mining the *Eventual* property on "normal" waveforms, the miner tracks the longest time observed between an a and b proposition and records it. When a mined *Eventual* property is checked on a failing waveform, every a proposition must be followed by a b within the observed bound. Note that the bounded *Eventual* property generalizes the *Next* property.

The mined LTL formulas are of the form $G(X \rightarrow Y)$ where X and Y are inner LTL formulas. The notion of *support* is defined to be the number of points on the trace where Y is satisfied, which is a specialization of the definition given by Lemieux et al. [8] Support measures how often a property has "completed", and is a heuristic that estimates the likelihood the property is true.

A property is *falsifiable* on a trace if X holds at any point in the trace and a property is *falsified* if $X \rightarrow Y$ is ever false. When merging properties mined from multiple waveforms, the miner discards any properties that were falsified on any waveform or were never falsifiable, and aggregates the support for properties that were never falsified.

2.3 Constructing Atomic Propositions

For the LTL templates in Section 2.1 to produce useful properties, the signals in the RTL design must be transformed into appropriate atomic propositions (APs). Firstly, we consider the signal values of 1-bit registers and wires in the design to be APs themselves. Secondly, we transform 1-bit signals into APs corresponding to the `$rose` and `$fell` system tasks in SystemVerilog, indicating whether the signal rose or fell from the last clock cycle. We also transform 1-bit and multi-bit signals into APs corresponding to the `$stable` system task in SystemVerilog, which indicates that the signal did not transition from the last clock cycle.

In a similar vein, Li et al. [9] convert traces of 1-bit and multi-bit signals into *delta traces*, where the delta trace is 1 if the signal transitioned on a given clock cycle and 0 otherwise. Figure 3 demonstrates the construction of delta traces from RTL signals.

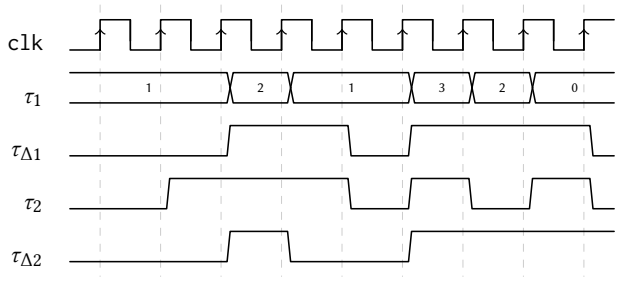


Figure 3: Examples of delta trace construction on a multi-bit (τ_1) and a 1-bit (τ_2) signal

LTL properties can then be mined over delta traces; this has the advantage of dealing with 1-bit and multi-bit signals uniformly and also takes advantage of the signal transition sparsity, common in digital designs, to build a fast spec miner.

However, delta events are insufficient to capture detailed behavior in multi-bit signals since they lose information about specific transitions. We additionally construct APs from multi-bit signals by creating a set of propositions that record transitions between every *pair of states* of the signal. This construction can be extended to produce APs that record transitions from an arbitrary state to a specific one, which is useful for capturing the reset behavior of FSMs. Figure 4 shows a trace of a 2-bit `state` register and AP traces constructed from specific transitions from state 1 to 2 and 3 to 0, as well as from an arbitrary state to 0.

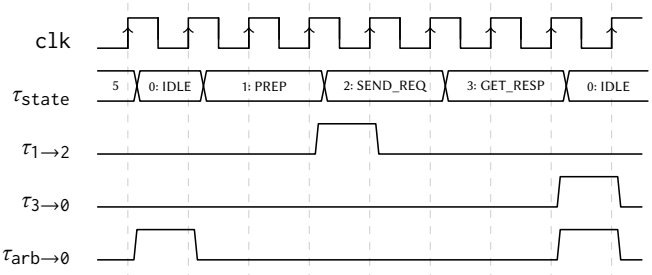


Figure 4: Examples of constructing transition APs on the multi-bit `state` signal.

Even though these proposition constructions produce tens of traces per multi-bit signal in the RTL design, they do not degrade the mining algorithm performance substantially since each produced trace still has sparsity in the number of transitions.

2.4 Extracting Design Semantics for Targeted APs

All the proposition constructions described in Section 2.3 are usable for any RTL design, but may not be specific enough to capture useful properties. We propose deriving more APs from automatic static analysis of the RTL and annotations passed down from the designer.

We take advantage of the FIRRTL compiler [5] to write analysis passes to detect common inter-module interfaces such as ready-valid or AXI4 and instantiated module types such as arbiters, FIFOs, and counters. This extracted metadata drives the construction of targeted APs for particular modules and interfaces. We present some targeted APs for two interfaces and two module types:

- Ready-Valid interface
  - Transaction fired: `ready` $\wedge$ `valid`
  - Backpressure applied: $\neg$`ready` $\wedge$ `valid`
- AXI-4 Bus
  - Multibeat write: `AWVALID` $\wedge$ `AWREADY` $\wedge$ `AWLEN` ≥ 0
  - Last write beat: `WVALID` $\wedge$ `WREADY` $\wedge$ `WLAST`
  - Specific read: `ARVALID` $\wedge$ `ARREADY` $\wedge$ `ARADDR` $==$ `ADDR`
- FIFO/Queue
  - Write while full (overflow): `wr_en` $\wedge$ `full` $\wedge \neg$`rd_en`
  - Write and read: `wr_en` $\wedge$ `rd_en`
- Counter
  - Overflow: `count` $==$ `COUNT_MAX` $\wedge$ `count_en`

Table 1: Properties mined on `riscv-mini` ranked by support

Template	$\tau_{\Delta a}$	$\tau_{\Delta b}$	Support
Until	`core.ctrl.io_A_sel`	`core.dpath.io_ctrl_imm_sel`	6113
Next	`core.dpath.brCond.eq`	`core.dpath.brCond.neq`	5457
Eventual	`core.dpath.io_ctrl_imm_sel`	`core.dpath.regFile_io_wen`	4466
Until	`icache.io_cpu_req_valid`	`icache.is_idle`	4359
Until	`icache.io_cpu_req_valid`	`icache.is_read`	4359
Eventual	`icache.io_cpu_req_valid`	`icache.is_read`	4336
Until	`icache.io_cpu_req_valid`	`core.dpath.csr_io_stall`	4318
Next	`core.dpath.csr.isEcall`	`dcache_io_cpu_abort`	45

– Reset: $\texttt{past(count)} \neq 0 \wedge \texttt{count} == 0$

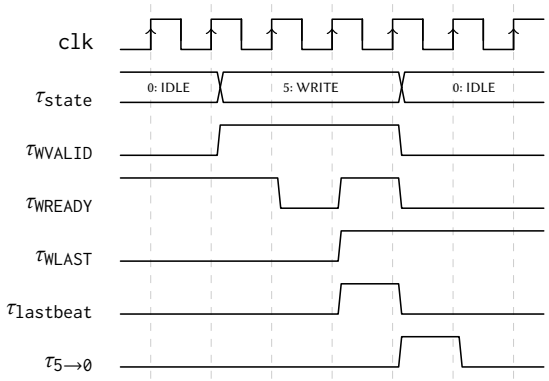


Figure 5: Combining state transition and AXI-4 specific APs to mine complex properties.

Figure 5 shows the behavior of an AXI4 master sending a 2 beat write transaction to a slave. $\tau_{\texttt{lastbeat}}$ is the trace of an AXI-4 specific AP constructed according to the 'Last write beat' formula above. The $\tau_{\texttt{lastbeat}}$ and $\tau_{5\to0}$ APs slot cleanly into the 'Next' LTL template, whereas separately examining the `WVALID`, `WREADY`, `WLAST`, and `state` signals would not have revealed this property.

2.5 Algorithm

Algorithm 1 Spec Miner

```
1:  procedure Miner([τ_1, ..., τ_N], M, S)
2:      P ← ∅
3:      for [τ_m], m ← Modularize([τ_{1,...,N}], M) do
4:          if ¬ IsLeaf(m) then
5:              [τ_m] ← StripInternal([τ_m])
6:          [τ_m] ← Trim([τ_m])
7:          [τ_{m,props}] ← ConstructProps(τ_m, S)
8:          for (τ_i, τ_j) ← Permutations([τ_{m,props}], 2) do
9:              for Checker ← LTL Templates do
10:                 P ← P ∪ Checker(τ_i, τ_j)
11:     return P
```

The spec mining algorithm is simple and fast to execute. The input is a value change dump (VCD) file which contains the traces $(\tau_1, \ldots, \tau_N)$ of the N signals in the RTL design (including both registers and named wires). The module hierarchy M is extracted from the VCD file and is used to partition the signals that can be combined together into a mined property.

For every module in the RTL design (`Modularize`), we only consider signals directly inside the module for mining ($[\tau_m]$), and furthermore, only the module's I/O ports are considered if the module is not a leaf of the hierarchy. To further constrain the mining algorithm, only signals 5 bits wide or less are considered (`Trim`); this strips away most datapath signals, leaving FSM state registers and control signals.

An optional input S describes the design semantics in each module for targeted construction of propositions (Section 2.4). The `ConstructProps` function uses S and the generic proposition construction formulas (Section 2.3) to build a set of traces of APs derived from $[\tau_m]$. Pairwise permutations of $[\tau_{m,props}]$ are substituted into each LTL template (Section 2.1) and each template's `Checker` returns a tuple containing the support, falsifiability, falsification, and liveness bounds for the candidiate traces plugged into the template. The return values of the `Checker` are aggregated and returned as the mined properties for the waveform.

2.6 HW Spec Mining Caveats

Spec mining on RTL traces has additional consideration when compared to mining on a software execution log.

2.6.1 Resets. RTL designs usually have one global reset and many sub-resets that can selectively reset a module or some part of its functionality. When mining, we do not want to falsify properties based on erroneous behavior which only occurs during a module's reset sequence, but not during regular execution.

To handle this, we extract global and module-level reset signals from static analysis of the RTL and only begin mining properties once the associated signals are out of reset. Additionally, we mine on RTL tests where the resets are exercised frequently to produce properties related to reset behavior.

2.6.2 Sampling. Digital circuits update state on one or more clock signals at the rising or falling clock edge. RTL static analysis is used to mark registers with their sampling clock and edge to handle sampling on multi-clock designs.

2.6.3 Trivial Properties. Many mined properties are uninteresting and trivial such as mining a *Next* property on delta traces entering and exiting a shift register chain. Some of these trivial properties

can be extracted from static RTL analysis and are used to skip over potential permutations during mining to improve runtime.

3 PRELIMINARY RESULTS

We apply our specification mining engine to `riscv-mini`[6], a simple RISC-V processor with caches written in Chisel. The processor contains about 200 signals that meet the criteria for mining. Specs were mined on VCDs produced by running all the ISA tests using *only* the delta trace proposition construction technique. After merging the mined properties from each waveform, a total of **3251** fine-grained specifications were produced, shown in Table 1. Note that properties with low support values can still be useful (e.g. `isEcall` → `cpu_abort`).

3.1 Bug Localization

After mining specs from a working version of `riscv-mini`, bugs were introduced in the source RTL. The injected bugs caused some ISA tests to fail and the failing waveforms were checked against the mined specs.

We introduced a typo bug in the control unit that changed how an illegal instruction was detected:

```
- io.illegal := ctrlSignals(12)
+ io.illegal := ctrlSignals(11)
```

Several ISA tests began to hang, and the failing VCDs were checked against the mined properties. The following properties were violated by the failing waveform:

$$\texttt{core.dpath.csr.io\_illegal}\ \mathbf{U}\ \texttt{icache.io\_cpu\_req\_valid}$$

$$\texttt{core.dpath.csr.io\_illegal}\ \mathbf{U}\ \texttt{icache.io\_cpu\_resp\_valid}$$

$$\texttt{core.dpath.csr.io\_illegal}\ \mathbf{U}\ \texttt{core.ctrl.io\_A\_sel}$$

The violated properties point to something wrong with the `io_illegal` signal, and effectively localize the bug.

In another example, we introduce a logic bug in the cache:

```
- hit := v(idx_reg) && rmeta.tag === tag_reg
+ hit := v(idx_reg) && rmeta.tag =/= tag_reg
```

This bug does not affect most ISA tests but causes a microbenchmark to fail by hanging. After checking the failing VCD against the mined properties, these violations were found, ranked by time of violation:

$$\texttt{arb\_io\_dcache\_r\_ready}\ \mathbf{U}\ \texttt{dcache.hit}\ (418)$$

$$\texttt{dcache\_io\_nasti\_r\_valid}\ \mathbf{U}\ \texttt{dcache.hit}\ (418)$$

$$\texttt{dcache.is\_alloc}\ \mathbf{U}\ \texttt{dcache.hit}\ (418)$$

$$\texttt{arb.io\_dcache\_ar\_ready}\ \mathbf{U}\ \texttt{arb\_io\_nasti\_r\_ready}\ (640)$$

The violated properties point to something wrong with the `hit` signal and localize the bug. Of note, this failing waveform had over 60 property failures, however the earliest failures point to the origin of the bug, while the subsequent failures indicate the downstream effects of the bug.

4 CONCLUSION AND FUTURE WORK

Our preliminary results show promise for using specification mining to localize RTL bugs by performing waveform anomaly detection. This work is being extended with the methods for constructing propositions and extracting design semantics explained in this paper. We are exploring techniques to detect bus protocols and interfaces from RTL analysis, extract transaction-level traces from VCDs, and use software spec miners to supplement fine-grained RTL-level properties. We are in the process of applying specification mining to localize difficult to diagnose bugs in more complex designs like BOOM [1].

ACKNOWLEDGMENTS

The information, data, or work presented herein was funded in part by the Advanced Research Projects Agency-Energy (ARPA-E), U.S. Department of Energy, under Award Number DE-AR0000849. Research was partially funded by ADEPT Lab industrial sponsor Intel ISTC, and ADEPT Lab affiliates Google, Siemens, SK Hynix, Apple, Futurewei, and Seagate. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

REFERENCES

[1] Christopher Celio, David A. Patterson, and Krste Asanovic. 2015. *The Berkeley Out-of-Order Machine (BOOM): An Industry-Competitive, Synthesizable, Parameterized RISC-V Processor.* Technical Report. https://www.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-167.html

[2] Harry D Foster. 2015. Trends in functional verification: A 2014 industry study. In *DAC*.

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

[4] Kai hui Chang, Ilya Wagner, Valeria Bertacco, and Igor L. Markov. 2007. Automatic error diagnosis and correction for RTL designs. In *2007 IEEE International High Level Design Validation and Test Workshop*. IEEE.

[5] Adam Izraelevitz, Jack Koenig, Patrick Li, Richard Lin, Angie Wang, Albert Magyar, Donggyu Kim, Colin Schmidt, Chick Markley, Jim Lawson, and Jonathan Bachrach. 2017. Reusability is FIRRTL ground: Hardware construction languages, compiler frameworks, and transformations. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE.

[6] Donggyu Kim. 2019. *Simple RISC-V 3-stage Pipeline in Chisel.* https://github.com/ucb-bar/riscv-mini

[7] Donggyu Kim, Christopher Celio, Sagar Karandikar, David Biancolin, Jonathan Bachrach, and Krste Asanovic. 2018. DESSERT: Debugging RTL Effectively with State Snapshotting for Error Replays across Trillions of Cycles. In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE.

[8] Caroline Lemieux and Ivan Beschastnikh. 2015. Investigating Program Behavior Using the Texada LTL Specifications Miner. In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE.

[9] Wenchao Li, Alessandro Forin, and Sanjit A. Seshia. 2010. Scalable specification mining for verification and diagnosis. In *Proceedings of the 47th Design Automation Conference on - DAC '10*. ACM Press.

[10] S. Mirzaeian, Feijun Zheng, and K.-T.T. Cheng. 2008. RTL Error Diagnosis Using a Word-Level SAT-Solver. In *2008 IEEE International Test Conference*. IEEE.

[11] A. Smith, A. Veneris, M.F. Ali, and A. Viglas. 2005. Fault diagnosis and logic debugging using Boolean satisfiability. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 24, 10 (oct 2005), 1606–1621.

[12] W. M. P. van der Aalst, H. T. de Beer, and B. F. van Dongen. 2005. Process Mining and Verification of Properties: An Approach Based on Temporal Logic. In *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 130–147.

[13] Shobha Vasudevan, David Sheridan, Sanjay Patel, David Tcheng, Bill Tuohy, and Daniel Johnson. 2010. GoldMine: Automatic assertion generation using data mining and static analysis. In *2010 Design, Automation & Test in Europe Conference & Exhibition (DATE 2010)*. IEEE.

[14] Jinlin Yang, David Evans, Deepali Bhardwaj, Thirumalesh Bhat, and Manuvir Das. 2006. Perracotta. In *Proceeding of the 28th international conference on Software engineering - ICSE '06*. ACM Press.