The Essence of Verilog

A Tractable and Tested Operational Semantics for Verilog

QINLIN CHEN, Nanjing University, China
NAIREN ZHANG, Nanjing University, China
JINPENG WANG, Nanjing University, China
TIAN TAN*, Nanjing University, China
CHANG XU, Nanjing University, China
XIAOXING MA, Nanjing University, China
YUE LI*, Nanjing University, China

With the increasing need to apply modern software techniques to hardware design, Verilog, the most popular Hardware Description Language (HDL), plays an infrastructure role. However, Verilog has several semantic pitfalls that often confuse software and hardware developers. Although prior research on formal semantics for Verilog exists, it is not comprehensive and has not fully addressed these issues. In this work, we present a novel scheme inspired by previous work on defining core languages for software languages like JavaScript and Python. Specifically, we define the formal semantics of Verilog using a core language called λ_V, which captures the essence of Verilog using as few language structures as possible. λ_V not only covers the most complete set of language features to date, but also addresses the aforementioned pitfalls. We implemented λ_V with about 27,000 lines of Java code, and comprehensively tested its totality and conformance with Verilog. As a reliable reference semantics, λ_V can detect semantic bugs in real-world Verilog simulators and expose ambiguities in Verilog's standard specification. Moreover, as a useful core language, λ_V has the potential to facilitate the development of tools such as a state-space explorer and a concolic execution tool for Verilog.

CCS Concepts: • **Software and its engineering** → **Semantics**; • **Hardware** → *Hardware description languages and compilation.*

Additional Key Words and Phrases: Verilog, Semantics, Hardware Description Languages, Core Languages

ACM Reference Format:
Qinlin Chen, Nairen Zhang, Jinpeng Wang, Tian Tan, Chang Xu, Xiaoxing Ma, and Yue Li. 2023. The Essence of Verilog: A Tractable and Tested Operational Semantics for Verilog. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 230 (October 2023), 30 pages. https://doi.org/10.1145/3622805

*Corresponding author.

Authors' addresses: Qinlin Chen, qinlinchen@smail.nju.edu.cn, State Key Laboratory for Novel Software Technology, Nanjing University, China; Nairen Zhang, nairen@smail.nju.edu.cn, State Key Laboratory for Novel Software Technology, Nanjing University, China; Jinpeng Wang, jpwang@smail.nju.edu.cn, State Key Laboratory for Novel Software Technology, Nanjing University, China; Tian Tan, tiantan@nju.edu.cn, State Key Laboratory for Novel Software Technology, Nanjing University, China; Chang Xu, changxu@nju.edu.cn, State Key Laboratory for Novel Software Technology, Nanjing University, China; Xiaoxing Ma, xxm@nju.edu.cn, State Key Laboratory for Novel Software Technology, Nanjing University, China; Yue Li, yueli@nju.edu.cn, State Key Laboratory for Novel Software Technology, Nanjing University, China.

2475-1421/2023/10-ART230
https://doi.org/10.1145/3622805

1 INTRODUCTION

With the emergence of domain-specific architectures (e.g., various neural network chips for deep learning), there is an increasing need to apply modern software techniques to hardware design to address issues of reliability, security, and productivity [Truong and Hanrahan 2019]. Verilog, the most popular Hardware Description Language (HDL) [Golson and Clark 2016], plays an infrastructure role in this trend. Verilog is designed to describe digital circuit models that can be synthesized into real hardware or simulated in software. Many hardware reliability and security methods [Grimm et al. 2018; Witharana et al. 2022] are developed upon Verilog, and new hardware languages like Chisel [Bachrach et al. 2012] that aim to improve hardware design productivity use Verilog as the back-end due to its mature industrial support for synthesis and simulation.

Despite its importance, Verilog has a number of semantic pitfalls, as outlined in Table 1. These pitfalls often arise from hardware-focused features that have unique but not well-designed semantics, which can confuse both software and hardware developers. For software developers who develop various tools for Verilog, these pitfalls can exhibit unusual behaviors that are difficult to handle compared to those of software languages. For hardware developers, understanding those pitfalls may be more challenging, as it often requires a deep understanding of software, e.g., simulators, making it hard to locate the root reasons of certain semantic bugs caused by the pitfalls. Moreover, these issues are exacerbated by the fact that these features are often described in inaccurate and ambiguous prose in Verilog's specification.

It is essential to emphasize that the semantics mentioned earlier encompass both hardware synthesis and simulation aspects. We argue that despite much Verilog code being purely synthesizable, it is crucial to consider the semantics for simulation. In the real world, hardware programmers dedicate substantial time to simulating, understanding, and debugging Verilog programs (even though SystemVerilog may be used by programmers to write testbenches for simulation, it still inherits the simulation mechanism from Verilog). Therefore, focusing solely on the synthesizable subset would be insufficient. As an example, delay control is frequently employed in simulation to sample outputs or generate inputs at specific times for RTL design under test. However, it often poses challenges for programmers in determining when exactly those delayed statements are executed, potentially leading to bugs. Formal specification of such features, even if they are not synthesizable, holds value in addressing these challenges. Notably, the recent prominent hardware representation work LLHD [Schuiki et al. 2020] also provides primitives to support behavioral features like delay controls, recognizing the significance of simulation scenarios.

Unfortunately, prior research on formal semantics for Verilog has fallen short in accurately and completely addressing these pitfalls in synthesis and simulation, as summarized in Table 2. For example, the state-of-the-art formal semantics for Verilog [Meredith et al. 2010] still fails to capture the essential difference between nets and variables, leading to the disregarding of the distinctive problem of multiple drivers in Verilog. Additionally, they did not account for a set of language features that are hard to extend based on its semantic core.

In this work, we aim to clarify those pitfalls by defining the formal semantics of Verilog using a core language λ_V that captures the essence of Verilog with as few language structures as possible. This core-language method has been used successfully in defining semantics for popular languages like JavaScript [Guha et al. 2010] and Python [Politz et al. 2013], and offers the benefits of facilitating proofs and tools, as well as providing insights into the language [Krishnamurthi 2015]. Our work is greatly inspired by these illuminating core-language works on software languages.

Through meticulous handling of these pitfalls, our λ_V language naturally yields the most complete formal semantics of Verilog to date. It covers nearly all features for synthesis and simulation, with exceptions to user-defined primitives, switch-level modeling, automatic functions and tasks, and

Table 1. Summarization of the representative Verilog features that have pitfalls in their semantics, which make previous semantic work inaccurate and often confuse developers. These features are categorized as per the Verilog specification. Section 2 provides an overview for a major portion of these features and Section 3 shows how our core language λ_V handles them formally. The relevant sections are listed in the final column, with "ext" indicating that the corresponding topic will be explored in our supplementary file.

Categories	Representative Features	Abbr.	Differences from Software Languages	Pitfalls in Semantics	Sections
Data	Bit Value	BV	Verilog's basic value is a 4-value bit: 0, 1, x, and z. The uncommon x and z bits represent unknown and high-impedance respectively.	Unexpected x and z values in evaluation.	3.1
	Net	N	Nets are designed to model physical wires. They are often used like variables but have unique ability to resolve problem of multiple drivers caused by continuous assignments.	Unaware of the resolution functions defined for nets to resolve the problem of multiple drivers.	3.2
Expression	Context-determined Expression	CE	The implicit type conversions in an expression are affected by the context where the expression is given.	Rules for such conversions are intricate and not clear in Verilog specification.	3.3
Timing Control	Delay Control Event Control Repeat Event Control	TC	Timing controls are synchronous primitives unique in Verilog, especially delay controls that are used to synchronize processes by waiting on time-related conditions.	Repeat event control is often mistakenly treated as syntax sugar. In fact, it requires extra states to be formally modeled.	3.4 & ext
Statement	Blocking Assignment	BA	Same as assignments in software languages.	None.	3.5.1
	Nonblocking Assignment	NBA	The RHS value is not assigned to LHS until the end of a time step. This unique semantics is to model the update to physical registers and can naturally avoid data-races in simulation.	Fail to ensure the determinism about the result of multiple NBAs to the same variable within a time step and fail to handle NBA with an optional TC.	3.5.2
	Procedural Continuous Assignment	PCA	Create and cancel a continuous assignment at runtime.	PCA can override plain continuous assignments, leading to complications in resolution functions.	ext
	Advanced Flow Control (e.g., fork, disable)	AFC	Resemble those used in software languages. Fork allows for creating concurrent executions at runtime, while the disable statement can break the control flows of processes.	Hard to extend existing works on Verilog semantics to support AFC.	ext
Continuous Assignment	Continuous Assignment	CA	The RHS's value is continuously assigned to the LHS, which is a Verilog-specific feature.	Multiple CAs to the same net or reg introduces the problem of multiple drivers. In addition, CA with TC has the inertial-delay problem.	3.6
Schedule	Scheduling Semantics	SCH	Determine how to schedule concurrent constructs in Verilog, such as continuous assignments and processes created by behavioral modeling.	Many implementations deviate from Verilog specification, resulting in inconsistent behaviors and possible hidden data races.	3.7
Module	Port Connection	CONN	Variables on two sides of input or output ports are connected using CAs.	For an inout port, both sides are aliased.	ext
Task&Function	Lifetime of Variables	LT	Functions and tasks in Verilog allow variables to be declared as either stack variables or static variables.	By default, variables are static.	ext
System T&F	System Task&Function	STF	Resemble system calls.	None.	ext
Gate-Level	Gate Instantiation	GATE	Model circuits using logical gates.	None.	ext

specific system tasks and functions. The rationales for exclusion are as follows: User-defined primitives, which were utilized in early Verilog days to form custom logic components based on truth tables, have become obsolete due to contemporary Verilog features that function equivalently. Switch-level modeling presents more elaborate hardware descriptions than logic gates, but its low-level nature may result in more complicated simulation. Furthermore, current synthesis tools can automatically optimize the circuits that they generate at this level. Automatic functions and tasks are uncommonly used in pure Verilog and are not synthesizable. Their primary purpose is to facilitate writing testbenches with the aid of advanced SystemVerilog features that extend beyond Verilog. System tasks and functions resemble system calls, and we only model the commonly used part of them.

Table 2. The completeness of related works on Verilog semantics. Features are from Table 1. One feature is marked by "○" if it is not supported or totally wrongly modeled by some work, by "●" if it is fully supported without mistakes, and otherwise by "◐", denoting partial support or (mostly) the pitfalls described in Table 1.

	BV	N	CE	TC	BA	NBA	PCA	AFC	CA	SCH	CONN	LT	STF	GATE
[Gordon 1995]	◐	◐	○	◐	●	◐	○	○	◐	◐	◐	○	○	◐
[Fiskio-Lasseter and Sabry 1999]	●	◐	○	◐	●	●	○	○	○	◐	◐	○	○	○
[He and Xu 2000]	◐	◐	○	◐	●	○	○	◐	○	◐	○	○	○	○
[He and Zhu 2000]	◐	◐	○	◐	●	○	○	◐	○	◐	○	○	○	○
[Dimitrov 2001]	◐	◐	○	◐	●	◐	○	○	◐	◐	○	○	○	○
[Meredith et al. 2010]	◐	◐	●	◐	●	●	○	○	◐	●	◐	○	◐	◐
Our work	●	●	●	●	●	●	●	●	●	●	●	◐	◐	●

Please note that λ_V is not so tiny as core languages designed for software languages because Verilog inherently surpasses the complexity of software languages. This complexity stems from its various timing controls, intricate scheduling mechanisms, unique procedural statements, such as nonblocking assignments, necessitating additional mechanisms to accurately and completely model its semantics. As a result, the core of λ_V is larger compared to software languages. We have balanced the tractability of λ_V while keeping its completeness by trying to minimize the number of constructs of λ_V, in the sense that certain Verilog features would not be fully modeled by λ_V if more λ_V constructs were further removed. As a result, we have desugared modules, port connections, functions, tasks, initial/always blocks, fork-join blocks, etc., into fewer and simpler constructs. Even though, it is nearly infeasible to present λ_V in full given the space limits. Thus, we have separately provided a supplementary file [Chen et al. 2023b] to formalize and explain all the other language features, highlighted as "ext" in Table 1, based on our semantic core.

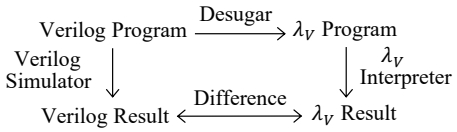


Fig. 1. Testing strategy for λ_V.

To demonstrate that λ_V has *totality* for desugaring real-world Verilog programs and has *conformance* with the Verilog specification, we implement and test it by following the strategy used in existing works for core languages [Guha et al. 2010; Krishnamurthi et al. 2019; Politz et al. 2013] as depicted in Figure 1. This strategy treats Verilog simulators as the closest approximation to the missing complete formal specification of Verilog and compares λ_V's results with those of real-world Verilog simulators on a sufficient number of test cases. For our comprehensive testing, we utilized two of the most popular open-source Verilog simulators: *Icarus Verilog* [Williams 2023] and *Verilator* [Snyder 2023a]. We also prepared two distinct suites of test cases to evaluate λ_V from different aspects: one suite consisting of 824 test cases sourced from the *Icarus Verilog* testbench, encompassing all representative features of Verilog, and the other suite comprising real-world programs primarily sourced from open-source projects. The evaluation results were promising, demonstrating that λ_V performs well in both totality and conformance; moreover, three types of real semantic bugs were uncovered in *Icarus Verilog* and *Verilator*, indicating that λ_V can be used as a reliable reference semantics for detecting semantic bugs in real-world simulators. In addition, our testing even exposed four types of ambiguities in Verilog's standard specification.

Finally, as a core language, the essence of λ_V lies in its minimal language constructs, which makes it easier to develop tools. In this regard, we discuss how λ_V and its interpreter can be utilized to develop other applications, such as a state-space explorer and a concolic execution tool for Verilog.

In summary, this work makes the following contributions:

- We define λ_V, the first core language for Verilog, which provides the most complete semantics to date, identifying and addressing previously overlooked pitfalls in existing literature.
- We provide an implementation of λ_V, comprising a desugaring translation from Verilog to λ_V and an interpreter for λ_V, encompassing approximately 27,000 lines of Java code (excluding comments).
- We demonstrate the totality and conformance of λ_V through comprehensive testing.
- We showcase λ_V's ability to detect semantic bugs in real-world Verilog simulators and expose ambiguities in Verilog's standard specification, making it a reliable reference semantics.
- We discuss the potential for λ_V to facilitate the development of a state-space explorer and a concolic execution tool for Verilog, highlighting its usefulness as a core language.
- We have provided an artifact [Chen et al. 2023a] that describes all the bugs and ambiguities detected, and we will make the implementations of λ_V fully open-source.

2 VERILOG, INFORMALLY

Verilog is widely recognized as the most popular hardware description language used for both *synthesizing* and *simulating* digital circuits [Golson and Clark 2016]. It allows developers to write programs that describe digital circuits, which can then be inputted into a *synthesizer* to create circuit-like representations for creating real hardware. This process, referred to as *synthesis*, is analogous to how a software compiler produces executables. Additionally, developers can use Verilog code to generate testing inputs for the hardware they have described. The entire program can then be run through a *simulator* to simulate the actual response of the hardware under those inputs. This *simulation* process is akin to program interpretation and is significantly less costly than testing the hardware in a real environment.

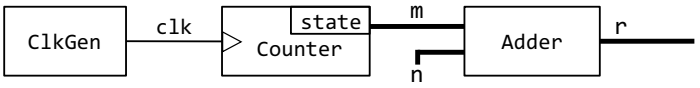
Verilog provides numerous features for synthesis and simulation. However, many of these features differ from those found in software languages, which can lead to pitfalls as outlined in Table 1. These pitfalls frequently perplex both hardware and software developers, make it challenging to achieve accuracy in prior semantic works on Verilog, and even result in semantic bugs in Verilog implementations, such as simulators. Before delving into λ_V that formally models Verilog semantics and handles the aforementioned pitfalls in Section 3, let us first use an introductory example to explain how some of the representative features of Verilog in Table 1 are employed to describe digital circuits for synthesis (Section 2.1) and how Verilog programs are run in simulation (Section 2.2).

2.1 Describe Digital Circuits

Figure 2 presents a simple example of a digital circuit and the corresponding Verilog code snippet describing it. The circuit comprises three electronic components interconnected by wires:

- Figure 2b: a clock generator that produces a clock signal `clk` that oscillates between `0` and `1` at regular intervals of physical time;
- Figure 2c: a counter that increments from 0 to 9 and resets to 0 when it reaches 9, whose count value `state` changes only when signal `clk` changes from `0` to `1`;
- Figure 2d: an adder that produces the circuit's output `r` by adding `m` (assigned from the count value `state` in Figure 2c) and the circuit's input `n`.

Verilog provides *nets* (`N`) to model the physical wires that connect components in the circuit. The four nets shown in Figure 2a are declared in Figures 2b and 2d using the syntax `wire` *id*. Nets can carry *bit values* (`BV`), which represent electronic signals flowing on them. By specifying an additional range, such as `wire [3:0] r`, a net can carry 4-bit values (with its wire *id* being `r` in this example).



(a) An example of a digital circuit.

```
1 reg c;
2 wire clk;
3
4 always #1 c = ~c;
5 assign clk = c;
```

(b) A clock generator.

```
1 reg [3:0] state;
2
3 always @(posedge clk)
4   if (state == 9) state <= 0;
5   else state <= state + 1;
```

(c) A synchronous counter.

```
1 wire [3:0] m;
2 wire [3:0] n;
3 wire [3:0] r;
4 assign m = state;
5 assign r = m + n;
```

(d) An adder.

Fig. 2. An example of a digital circuit described in Verilog that covers many key language features.

Verilog often employs high-level features to describe the logic between input and output wires of components, rather than explicitly describing the *logic gates* (GATE) that make up the components. These features are classified into two categories based on whether the logic is combinational or sequential. Briefly, *combinational logic* has no state, while *sequential logic* accumulates a state based on past inputs.

The adder (Figure 2d) is a typical combinational logic circuit, whose output is the sum of its inputs. To describe it, a *continuous assignment* (CA) "`assign r = m + n`" is used. This means that the value of `r` changes immediately whenever there is a change in either `m` or `n`, just like they are physically connected and affected. The logic between the inputs and outputs (i.e., add the values of `m` and `n` to `r`) is also intuitively modeled by applying the operation of "`+`".

The counter (Figure 2c) is a typical sequential logic circuit. Sequential logic is essentially a state machine, and Verilog provides variables and imperative statements to describe its states and state transitions. In the example, a variable `state` is declared as `reg [3:0]` to track the current count, and the statements are grouped into an `always` block to describe the count increasing until it reaches 9, at which point it resets back to 0 and continues counting indefinitely (by `always`'s semantics).

It is important to note the difference between *nonblocking assignments* (NBA) and *blocking assignments* (BA). Nonblocking assignments were originally created to simulate the assignment to a register, which takes effect when the clock reaches positive/negative. Thus a nonblocking assignment, such as "`state <= state + 1`," is executed at a uniformly specified time step, while a blocking assignment, like "`state = state + 1`," is not. In our example, we use nonblocking assignments to prevent data races. If we used blocking assignments instead, it would be unclear whether the old or new value of `state` would be read when another sequential logic concurrently accessed its value. With nonblocking assignments, the new value of `state` is not immediately assigned, but rather at a certain time step (will be explained later), when all other concurrent sequential logic has read the old value. This ensures that data races are avoided, and the code's behavior becomes deterministic.

Moreover, the counter is a synchronous circuit that only updates its state when a global clock changes its value. Verilog provides *event control* (TC) "`@(posedge clk)`" (see Figure 2c) to achieve this behavior, which blocks state transitions until the clock signal flips from `0` to `1` (specified by the positive edge of the clock, a.k.a., `posedge`).

The clock generator (Figure 2b) is the first component in the example circuit and has a unique feature: it outputs an oscillating signal based on physical time. In Verilog, physical time is simulated

using steps instead of units like microseconds, and *delay controls* (also denoted as TC) "#n" are provided to block sequential logic for n time steps. The clock generator, as shown in Figure 2b, is implemented by flipping its state c every one time step (#1) and driving that value to the wire clk, which is used by other components such as the counter.

2.2 Run in Simulation

To simulate the Verilog code in Figure 2, we need to provide input to the circuit. This can be achieved using the code shown in Figure 3. The input generation code initializes the input n, clock flip variable c, and state and blocks itself for zero time step (#0). It then prints the output using the Verilog system task $display to check the value of output r. After two time steps (#2), the output value is displayed again. The main code is wrapped in an initial block that executes the statements in its body only once during simulation. On the other hand, an always block (refer to the one in Figure 2c) creates a process that executes its statements infinitely.

```
1 assign n = 1
2 initial begin
3   c = 0; state = 0; #0;
4   $display("%d", r);
5   #2 $display("%d", r);
6 end
```

Fig. 3. The example code to generate inputs for the code in Figure 2

While the code is simple, based on our experience, many Verilog programmers are unclear about how it will be executed due to the intricacies of Verilog's scheduling semantics in its specification. To facilitate comprehension of the subsequent material, we provide an intuitive overview of how Verilog programs are simulated, with an explanation of the usage scenarios behind each phase.

Basically, Verilog schedules the concurrent execution of continuous assignments (each of which can be seen as a statement executed in a separate process) and processes created from initial and always blocks. For simplicity, we focus on the schedule of processes and assume that whenever the right-hand side value of a continuous assignment changes, the new value is immediately assigned to the left-hand side. Consequently, the scheduler operates as follows:

(1) All processes run concurrently until they are blocked by event or delay controls. Sequential logic typically waits for a change in the clock signal (such as `@(posedge clk)` in Figure 2c), while combinational logic waits for input changes (such as m and n in Figure 2d). If any control conditions in process P are activated during the execution of these processes, P is scheduled to run concurrently with those processes.
(2) When all processes are blocked, scheduler will activate any processes that have zero delay (#0) and move to Phase (1). They are often test input generators that purposely wait before generating the next input or printing results, to ensure the circuit fully reacts to their inputs.
(3) If there are no zero-delay processes to execute, any pending non-blocking assignments are executed (which can be used to avoid data race as discussed earlier). If these assignments activate any processes, the scheduler returns to Phase (1). This moment (the point of time before executing Phase (4)) is referred to as the end of a time step since the scheduler will advance the time step in the next phase.
(4) All processes are blocked, and the simulation time advances by one step. This activates any delayed processes that were scheduled to run at that time. This typically results in clock generator flipping its output, which then activates any sequential logic waiting on the clock. Subsequently, the scheduler returns to Phase (1). This scheduling strategy ensures that the clock does not flip its value until all other sequential logic completed their state transitions.

Let us use the example in Figure 4 to illustrate this simulation process. We suggest following this figure while referring to the scheduling phases explained earlier and the code in Figures 2 and 3.

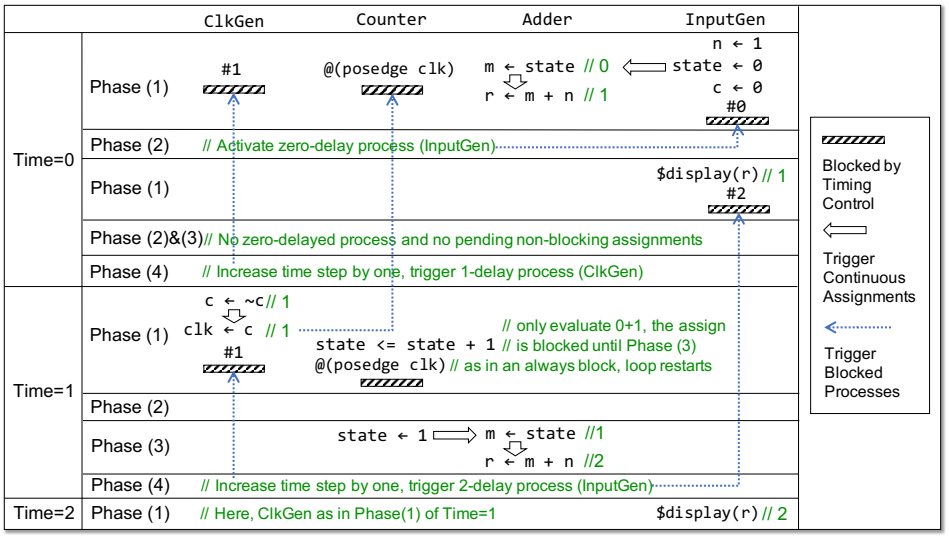


Fig. 4. An illustration of the simulation (execution) process for the code in Figures 2 and 3.

Figure 4 displays four columns `ClkGen`, `Counter`, `Adder` and `InputGen`, representing the four processes created for the code in Figures 2b,2c,2d, and Figure 3, respectively. These processes run concurrently, and for ease of illustration, we will focus on the flow that starts from `InputGen`.

During Phase (1) of Time 0, `state` is initialized to 0, triggering the continuous assignment `m = state` in `Adder`. As a result, `m` is immediately updated, which further triggers the next continuous assignment, updating `r` by `m + n`. Finally, `InputGen` is blocked by executing `#0`, which is a delay control. At this point, all processes are blocked, and following Phase (2), `InputGen` is activated since it contains zero-delay. This triggers the `$display` on line 4 in `InputGen`, outputting 1 for `r` (= `m + n` = 0 + 1). After this, the process is blocked again by executing `#2`. Since there are no zero-delayed processes and no pending non-blocking assignments, the current time step is passed, as described in Phase (3).

Moving on to Phase (4), the time step advances by one, activating `ClkGen`. Here, `c` is flipped because delay `#1` is reached, and its change further triggers a continuous assignment to `clk`. However, when `clk` changes from 0 to 1 as described in Phase (1), it immediately triggers the `posedge` of `Counter`. As a consequence, the non-blocking assignment `state <= state + 1` is executed by the `Counter`. However, any non-blocking assignment will be blocked until the end of the current time step, which is Phase (3). As a result, the non-blocking assignment is activated during Phase (3), and the value of `state` is assigned 1 (= 0 + 1). This immediately triggers the continuous assignments in `Adder`, and as a result, `r` is assigned 2 (= 1 + 1) finally.

We have now arrived at Phase (4) once again, and the time step has advanced by one, bringing us to time step 2. This triggers any delayed processes that were scheduled to run at this time, according to Phase (4) (which is identified as `#2` in our scenario). As a result, the `$display` function after `#2` in `InputGen` is executed, resulting in the final output of 2 for `r`.

3 λ_V: A TRACTABLE SEMANTICS FOR VERILOG

λ_V is a core language that is as expressive as Verilog yet keeps tractable for proofs and tools. In this section, we show how λ_V addresses the representative features summarized in Table 1, with a focus on those that are prone to pitfalls, by explaining the desugaring process or the formal operational

p	$\in$ *Program*	:=	$\epsilon \mid \iota; p$	c_{ev}	$\in$ *EventCtrl*	:=	`@(`η`)`
ι	$\in$ *Item*	:=	$k\ \ id : \tau \mid$ `var` $id : \tau \mid$	c_d	$\in$ *DelayCtrl*	:=	`#`e
			`assign` $c_d?\ \ id = e \mid$	η	$\in$ *EventExpr*	:=	$e \mid$ `posedge` $e \mid$
			`proc` s				`negedge` $e \mid \eta$ `or` η
k	$\in$ *NetKind*	:=	`wire` | `wor` | `wand` | $\cdots$	s	$\in$ *Stmt*	:=	`skip` | $s; s \mid s \parallel s \mid$
id	$\in$ *Id*	:=	{(symbols)}				`if` e `then` s `else` $s \mid$
τ	$\in$ *Type*	:=	`b1` | `b2` | $\cdots$				`while` e `then` $s \mid$
bv	$\in$ *BitVec*	:=	{(bit vectors)}				$id = c?\ e \mid id \Leftarrow c?\ e \mid$
e	$\in$ *Expr*	:=	$bv \mid id \mid op_m(e_1, \ldots, e_m)$				$c\ s \mid$ `blocked` pid
c	$\in$ *TimingCtrl*	:=	$c_{ev} \mid c_d$	pid	$\in$ *ProcId*	:=	{(symbols)}

Fig. 5. The syntax of λ_V, which uses `fixed-width` fonts for terminals and *italics* for non-terminals. The symbol $x?$ means x is optional. Despite its name, λ_V follows the style of Verilog rather than the λ calculus. This decision was made to facilitate the transfer of insights gained from λ_V to Verilog.

semantics. Due to space limitations and readability concerns, we only present a subset of λ_V that covers the most commonly used features (and explain the remaining ones in the supplementary material). However, this does not imply that other features are less important. For instance, Verilog provides modules to enable the reuse of hardware description code, and these modules have ports that are comparable to parameters in software functions. A special port of type `inout` provides the ability to model buses, which are common in hardware. Because the specification only mentions that it is "bidirectional" without any further explanation, all previous work miss its semantics, but we discovered its alias semantics through an error when we intuitively (but incorrectly) modeled real-world Verilog bus code by treating "bidirectional" as two mutual continuous assignments. In our supplementary material, we provide detailed explanations of those features and present the complete formal syntax and semantics of λ_V, which we refer to as Full-λ_V in the following sections.

The syntax and operational semantics of λ_V are formally defined in a series of figures. Figure 5 presents the syntax of λ_V. A λ_V program p consists of a list of items, which correspond to the items constituting Verilog modules. We define four kinds of items in λ_V:

- Net items: $k\ \ id : \tau$, desugared from Verilog's net declarations.
- Variable items: `var` $id : \tau$, desugared from variable declarations in Verilog.
- Assign items: `assign` $c_d?\ \ id = e$, directly correspond to Verilog's continuous assignments.
- Process items: `proc` s, used to create processes executing statements s concurrently. Initial blocks and always blocks in Verilog are desugared to `proc` s and `proc while 1 then` s.

Figure 6 defines the configuration, which represents the runtime state of a λ_V program, and defines function preprocess(p), which specifies how a λ_V program p is transformed into an initial configuration *conf*. We will further explain the rules in Figure 6 in the relevant sections below. Semantic rules for timing controls, statements, and continuous assignments are provided in Figures 7, 8, and 9. Figure 10 presents the scheduling rules. In the rest of this section, we follow the order of Verilog features in Table 1 to explain their semantics in λ_V.

General Notations. The $\sigma, \beta, \kappa, \alpha, u_\alpha$ in the definition of configuration (Figure 6) are all maps that can map a key k to a value using the notation like $\sigma(k)$. We denote the key set and value set of a map m as keys(m) and vals(m), respectively. We define $m(x) = \bot$ if and only if $x \notin$ keys(m). Here, $\bot$ can be interpreted as the `null` in programming languages. We use $\{\}$ to represent an empty map, and extending a map m by a new key-value pair (k, v) is represented as $m[k \mapsto v]$, whose formal definition is $m[k \mapsto v](k') = (k' = k)\ ?\ v : m(k)$. We also treat a map as a set of key-value mappings like $\{k \mapsto v, \ldots\}$. For convenience, we use notation $\square_\bot$ to represent $\square \cup \{\bot\}$, and use Σ to represent all but the first element of *conf* in the following sections, i.e., $\Sigma := \sigma, \beta, u_{nb}, \kappa, \alpha, u_\alpha, t$.

$$conf := (s, \sigma, \beta, u_{nb}, \kappa, \alpha, u_\alpha, t),$$

$s \in Stmt$	the statement being executed
$\sigma : (Id \cup AssignId) \mapsto BitVec$	the store that maps variables, nets, and assign items to their values
$\beta : ProcId \mapsto Cond$	a map from blocked processes to their waiting conditions
$u_{nb} \subseteq (Id \times BitVec \times Cond)^*$	a list of update events ($Id \times BitVec$) controlled by conditions ($Cond$)
$\kappa : Id \mapsto NetKind$	a map from nets to their net kinds
$\alpha : AssignId \mapsto (Id \times Expr \times DelayCtrl_\perp)$	a map from assign items to their runtime information
$u_\alpha : AssignId \mapsto (Id \times BitVec \times Cond)$	a map from assign items to their update events and control conditions
$t \in \mathbb{N}$	the simulation time step
$AssignId := \{(\text{symbols})\}$	the domain of identifiers for assign items
$Cond := EventExpr \cup \mathbb{N} \cup \{\checkmark\}$	the domain of blocking conditions

$$\text{preprocess}(p) = conf \quad \text{iff} \quad p, (\texttt{skip}, \{\}, \{\}, \epsilon, \{\}, \{\}, \{\}, 0) \rightarrow_{pp}^* \epsilon, conf.$$

$$\text{(PP-Net)}\ \frac{\sigma' = \sigma[id \mapsto \text{defVal}(k, \tau)] \quad \kappa' = \kappa[id \mapsto k]}{k\ id : \tau; p, (\ldots, \sigma, \ldots, \kappa, \ldots) \rightarrow_{pp} p, (\ldots, \sigma', \ldots, \kappa', \ldots)} \qquad \text{(PP-Var)}\ \frac{\sigma' = \sigma[id \mapsto \text{defVal}(\texttt{var}, \tau)]}{\texttt{var}\ \ id : \tau; p, (\ldots, \sigma, \ldots) \rightarrow_{pp} p, (\ldots, \sigma', \ldots)}$$

$$\text{(PP-CCA)}\ \frac{aid = \text{newId}() \quad \alpha' = \alpha[aid \mapsto (id, e, c_d)] \quad u'_\alpha = u_\alpha[aid \mapsto \text{genUpdEv}((id, e, c_d), \sigma, t)]}{\texttt{assign}\ \ c_d\ \ id = e; p, (\ldots, \alpha, \ldots, u_\alpha, \ldots) \rightarrow_{pp} p, (\ldots, \alpha', \ldots, u'_\alpha, \ldots)}$$

$$\text{(PP-CA)}\ \frac{aid = \text{newId}() \quad \alpha' = \alpha[aid \mapsto (id, e, \perp)] \quad u'_\alpha = u_\alpha[aid \mapsto \text{genUpdEv}((id, e, \perp), \sigma, t)]}{\texttt{assign}\ \ id = e; p, (\ldots, \alpha, \ldots, u_\alpha, \ldots) \rightarrow_{pp} p, (\ldots, \alpha', \ldots, u'_\alpha, \ldots)}$$

$$\text{(PP-Proc)}\ \frac{}{\texttt{proc}\ s; p, (s', \ldots) \rightarrow_{pp} p, (s \parallel s', \ldots)}$$

Fig. 6. Definition of configurations that represent the runtime states of λ_V, and the preprocess(p) function that transforms a λ_V program p into an initial configuration $conf$. The function is defined using a reduction relation $\iota; p, conf \rightarrow_{pp} p, conf'$ that processes each item in the program and updates the configuration accordingly.

We denote reduction relation between runtime configurations by $conf \rightarrow conf'$, and for simplicity, we omit unchanged elements of $conf$ in semantic rules.

3.1 Value and Type

Verilog values are represented by bit vectors bv in λ_V, as defined in Figure 5. Correspondingly, the only type τ defined in the language is the bn type, which represents fixed-width n-bit vectors.

Unlike in software languages, bit vectors in Verilog consist of four-value bits: `0`, `1`, `x`, and `z`. The `x` value represents an unknown bit and can occur when a bit vector is accessed by an index out of its range. The `z` value represents high impedance and indicates the absence of signals. The computation for bit vectors become more complicated when `x` and `z` values are involved, and we will present a real case that produces unexpected `x`/`z` results for bit vectors in Section 5.2.

While theoretically bit vectors are able to model any data, λ_V can be extended with additional data types for pragmatic convenience, e.g., Full-λ_V includes real numbers and arrays.

3.2 Nets and Variables

Nets and variables are fundamental components in Verilog that respectively model physical connections and represent states. In λ_V, they are declared using the two items: the net item $k\ \ id : \tau$ and the variable item `var` $\ id : \tau$, as shown in Figure 5. The id represents identifiers of the declared nets and variables. The type τ restricts possible values a net or variable can hold. The net kind k includes the commonly used `wire`, as well as other kinds discussed later. Note that we adopt the term *net kind* for what is called the *net type* in Verilog to avoid confusion because a net type (e.g.,

`wire`) plays no role as types that are traditionally used to restrict what values a net can hold. Note that the syntax of variable declaration "`reg a`" in Verilog is replaced by a clearer "`var a:b1`" in λ_V, because the keyword `reg` is misleading as these variables are not necessarily synthesized to be physical registers.

Net items and variable items are desugared from Verilog's net and variable declarations. For example, "`wire [2:0] u`" in Verilog corresponds to "`wire u:b3`" in λ_V. The difference is that range declarations (e.g., `[2:0]`) are replaced by type annotations (e.g., `b3`). In Verilog, the range declaration serves two purposes: indicating that the net has bit vector values and specifying how bits are indexed in the net, i.e., bits of a net declared with range `[h:l]` (e.g., `[4:2]`) can be accessed by indices l, $l+1$, ..., h (e.g., 2, 3, and 4). In λ_V, the latter feature is desugared to `b`n ($n = h - l + 1$) vectors, and all indices i of the range in Verilog are replaced by $i - l$. Therefore, λ_V only uses one kind of type annotations for all range declarations.

When preprocessing net and variable items, λ_V allocates space in the store σ for the nets and variables, and initializes them with default values given by function $\mathrm{defVal}(k, \tau)$, as formalized by the rules PP-Net and PP-Var in Figure 6. Full-λ_V provides the definition of $\mathrm{defVal}(k, \tau)$.

Difference between Nets and Variables. The syntactic difference between nets and variables is well-known: nets can only be used for continuous assignments, while variables are limited to procedural assignments. This restriction prevents inadvertent misuse of nets and variables in different assignments. However, the semantic difference between nets and variables are not well-understood. For example, [Meredith et al. 2010] concludes that nets and variables are indistinguishable at runtime when they both represent values, which is not true.

Thanks to the development of λ_V, we find that the essential difference between nets and variables lies in their handling of the *multiple drivers problem*. This problem occurs when a net has multiple continuous assignments driving it, making it unclear what value the net should take. Such scenarios often arise when modeling *bus*, which is a special net that have multiple electronic components connecting to it, driving or receiving signals from it. Without properly addressing the multiple drivers problem, such scenarios cannot be modeled. Note that variables can also have multiple drivers problems because Verilog provides a statement called procedural continuous assignment (`PCA`) that attaches (or detaches) a continuous assignment to a variable at runtime, which is intentionally used to force the variable to hold some value for a certain period in simulation.

Previous work on Verilog semantics overlooked the multiple drivers problem, as shown in Table 2, resulting in a incorrect model for nets, e.g., [Meredith et al. 2010] as discussed above.

To handle multiple drivers, we use *resolution functions* in λ_V to merge conflicting values from those drivers and take the results as their values. Resolution functions are specified by net kinds. For instance, if a net is declared as "`wor u:b3`" (the wire OR kind) and is driven by two continuous assignments, "`assign u = 3'b010; assign u = 3'b001`", it would apply the "or" function to the conflicting values `3'b010` and `3'b001` and set its final value as `3'b011`. If the net is declared as the common `wire` kind, according to the specification, its driving values are resolved to `3'b0xx`, where `x` represents an unknown bit. This process is formalize as $bv_r = \mathrm{resolve}(id, \sigma', \kappa, \alpha)$ by the rule E-ContUpdate in Figure 9, where id identifies a net that have multiple drivers, σ' and α are used to find those driving values, and κ records the net kind used to look up the resolution function for each net. The semantics of the resolve function is explained in details in Section 3.6.

In contrast, variables do not have resolution functions. When several continuous assignments are attached to the same variable by several `PCA`s, the latest continuous assignment will simply overwrite the previous ones. Those semantics are formally defined in Full-λ_V.

3.3 Expressions

Figure 5 gives the syntax of expressions, which is defined recursively by applying an m-ary primitive operator op_m to m sub-expressions, with bit vectors and identifiers serving as the base cases. We define the evaluation of an expression e with respect to a store σ using the function $[\![e]\!]\sigma$:

$$[\![e]\!]\sigma = \begin{cases} bv & e = bv \\ \sigma(id) & e = id \\ \delta_m(\mathrm{op}_m, [\![e_1]\!]\sigma, \ldots, [\![e_m]\!]\sigma) & e = \mathrm{op}_m(e_1, \ldots, e_m) \end{cases}$$

Here, we use an auxiliary function δ_m to model the semantics of all primitive operators op_m. For a full list of operators and the definition of δ_m, please refer to our supplementary material. Note that some Verilog expressions can have side effects on the store σ, so we replace $[\![e]\!]\sigma$ by a more complicated reduction relation $e, \Sigma \longrightarrow e', \Sigma'$ in Full-λ_V.

Context-Determined Expressions. In Verilog, expressions are weakly typed and thus their evaluations involve implicit type conversions, which is an error-prone feature. In addition, Verilog expressions are context-determined. This means that the implicit type conversions are not only determined by the expressions themselves, but also determined by the context in which the expressions are used. For example, consider the expression `a + b` in an assignment `c = a + b`, where `a` and `b` are unsigned 32-bit vectors and `c` is an unsigned 33-bit vector. According to Verilog specification [ver 2006], `c` is the context of `a + b` and determines what we call the *context type* of `a` and `b`, which is an unsigned 33-bit vector type. Therefore, `a` and `b` will be converted to unsigned 33-bit vectors to match their context types before being added. If `c` becomes a 34-bit vector, `a` and `b` will be converted to 34-bit vectors accordingly.

Context-determined expressions may have some benefits in certain scenarios, such as capturing overflowed bits in addition. However, they exacerbate the issues associated with implicit type conversions. Unfortunately, Verilog specification only provides an intricate and unclear description of context-determined expressions, leaving them as pitfalls as shown in Table 1. As a result, they are difficult for developers to understand and thus error-prone. For example, *Icarus Verilog* [Williams 2023], a popular Verilog simulator, could produced false results on context-determined expressions, as we explore in Section 5.1.1.

In λ_V, the implicit type conversions in context-determined expressions are made explicit during desugaring (we carefully read Verilog specification and resolve inaccuracy in description of context-determined expressions). For example, the add expression in the example above is desugared as `add(zext(a, 33), zext(b, 33))` in λ_V, where `zext` is a λ_V primitive operator that extends the operand by zeros to the target length.

3.4 Timing Controls

Timing controls are critical synchronous primitives in Verilog, with three types: delay controls, event controls, and repeat event controls. While λ_V supports all three types, due to space constraints, we do not introduce the less commonly used repeat event controls, but the semantic rules given in this section can be easily extended to support them. For more information, please refer to our supplementary material.

We explain the semantics of timing controls under the help of its typical usage as to block processes (related rules are listed in Figure 7). When a process executes a statement guarded by a timing control, it is blocked until the condition specified by the timing control is satisfied. The code snippet in Figure 2 has illustrated such usage. Recall that the delay control "`#1`" in Figure 2b blocks the clock generator from flipping its output until the simulation time advances one step, and the event control "`@(posedge clk)`" in Figure 2c blocks the execution of sequential logic until `clk` changes from `0` to `1` .

$$\frac{}{@(\eta), \sigma, t \Downarrow_c \eta, \sigma, t}\ (\text{E-EvCtrl}) \qquad \frac{[\![e]\!]\sigma = bv \quad \triangle = (\mathbf{x}|\mathbf{z} \in bv)\ ?\ 0 : \text{uint}(bv)}{\#e, \sigma, t \Downarrow_c (\triangle + t), \sigma, t}\ (\text{E-DlCtrl})$$

$\text{uint}(bv)$ = interpret bv as a two's complement unsigned integer
$\text{int}(bv)$ = interpret bv as a two's complement integer

$$\frac{c, \sigma, t \Downarrow_c cond, \sigma, t \quad pid = \text{newId}() \quad \beta' = \beta[pid \mapsto cond]}{c\ s, \beta \rightarrow (\texttt{blocked}\ \ pid; s), \beta'}\ (\text{E-Block}) \qquad \frac{\beta(pid) = \checkmark}{\texttt{blocked}\ \ pid \rightarrow \texttt{skip}}\ (\text{E-Activate})$$

Fig. 7. Semantic rules for establishing conditions from timing controls using $c, \sigma, t \Downarrow_c cond, \sigma, t$, along with rules for the primary usage of timing controls to block process execution ($c\ s$). Unchanged elements in configurations on both sides of $\rightarrow$ are omitted for clarity.

As shown in Figure 5, in λ_V, a timing control is denoted by c, and a statement guarded by it is denoted by $c\ s$. To model the semantics of $c\ s$, λ_V first establishes the condition $cond$ specified by the timing control using the reduction relation $c, \sigma, t \Downarrow_c cond, \sigma, t$, where σ and t are used to compute the condition. For an event control $@(\eta)$, its condition is its event expression η (by E-EvCtrl), and for a delay control $\#e$, its condition is the delay value evaluated from e (by E-DlCtrl) plus the current time (i.e., the t in configuration).

When a condition $cond$ is established, we block the process executing $c\ s$ by E-Block. Specifically, the process is blocked by a special statement $\texttt{blocked}\ \ pid$ before executing s. In addition, λ_V puts a unique pid along with the condition $cond$ to the β.

Now we explain how a blocked process is activated. This happens when the blocking condition becomes satisfied (denoted by $\checkmark$) on a certain event. Conditions may become satisfied on two kinds of events: *store changed* events, represented by a pair (σ, σ') that records the old store and the new store, and *time advancing* events, represented by the time step t. A store changed event is generated when a value in the store changes (Sections 3.5), and a time advancing event occurs when the simulation time advances (Sections 3.7). Once an event is generated, a helper function onEvent($cond, ev$) will be used to judge whether a condition is satisfied on the new event:

$$\text{onEvent}(cond, ev) = \begin{cases} \checkmark & cond = \eta \wedge ev = (\sigma, \sigma') \wedge \text{isRelevant}(\eta, \sigma, \sigma') \\ \checkmark & cond = t_\triangle \wedge ev = t \wedge t_\triangle \leq t \\ cond & \text{otherwise} \end{cases}$$

The isRelevant(η, σ, σ') is used to check whether the store change event (σ, σ') can satisfy η. The reason for this is that onEvent() is called for η on every store change, but many of these changes are irrelevant to η. The definition of isRelevant() is as follows:

$$\text{isRelevant}(\eta, \sigma, \sigma') = \begin{cases} \text{isRelevant}(\eta_1, \sigma, \sigma') \vee \text{isRelevant}(\eta_2, \sigma, \sigma') & \eta = \eta_1\ \ \texttt{or}\ \ \eta_2 \\ (\text{lsb}([\![e]\!]\sigma), \text{lsb}([\![e]\!]\sigma')) \in E_p & \eta = \texttt{posedge}\ \ e \\ (\text{lsb}([\![e]\!]\sigma), \text{lsb}([\![e]\!]\sigma')) \in E_n & \eta = \texttt{negedge}\ \ e \\ [\![e]\!]\sigma \neq [\![e]\!]\sigma' & \eta = e \end{cases}$$

where lsb(bv) returns the least significant bit of the bit vector bv, $[\![e]\!]\sigma$ denotes the evaluation result of expression e in store σ, and E_p and E_n are edge sets that define all possible posedges and negedges. For four-value bits, $E_p = \{(\texttt{0}, \texttt{1}), (\texttt{0}, \texttt{x}), (\texttt{0}, \texttt{z}), (\texttt{x}, \texttt{1}), (\texttt{z}, \texttt{1})\}$ and $E_n = \{(\texttt{1}, \texttt{0}), (\texttt{1}, \texttt{x}), (\texttt{1}, \texttt{z}), (\texttt{x}, \texttt{0}), (\texttt{z}, \texttt{0})\}$. If the associated condition of a process pid in β is satisfied by a new event, $\beta(pid)$ will be updated to $\checkmark$ (as explained in Sections 3.5 and 3.7). Subsequently, as per E-Activate, process pid resumes execution.

3.5 Statements

In this section, we introduce how λ_V models the statements in initial and always blocks in Verilog to describe hardware components. Production of *Stmt* in Figure 5 defines the syntax of λ_V statements.

$$\frac{s_1, \Sigma \to s_1', \Sigma'}{s_1; s_2, \Sigma \to s_1'; s2, \Sigma'}\ (\text{E-Seq}) \qquad \frac{s_1, \Sigma \to s_1', \Sigma'}{s_1 \parallel s_2, \Sigma \to s_1' \parallel s_2, \Sigma'}\ (\text{E-Par-L}) \qquad \frac{s_2, \Sigma \to s_2', \Sigma'}{s_1 \parallel s_2, \Sigma \to s_1 \parallel s_2', \Sigma'}\ (\text{E-Par-R})$$

$$\frac{}{\texttt{skip}; s \to s}\ (\text{E-Seq-Elim}) \qquad \frac{}{\texttt{skip} \parallel s \to s}\ (\text{E-Par-ElimL}) \qquad \frac{}{s \parallel \texttt{skip} \to s}\ (\text{E-Par-ElimR})$$

$$\frac{[\![e]\!]\sigma = bv \quad \neg(\mathbf{x}|\mathbf{z} \in bv) \wedge bv \neq 0}{\texttt{if } e \texttt{ then } s_1 \texttt{ else } s_2 \to s_1}\ (\text{E-If-True}) \qquad \frac{[\![e]\!]\sigma = bv \quad (\mathbf{x}|\mathbf{z} \in bv) \vee bv = 0}{\texttt{if } e \texttt{ then } s_1 \texttt{ else } s_2 \to s_2}\ (\text{E-If-False})$$

$$\frac{}{\texttt{while } e \texttt{ then } s \to \texttt{if } e \texttt{ then } (s; \texttt{while } e \texttt{ then } s) \texttt{ else skip}}\ (\text{E-While})$$

$$\frac{[\![e]\!]\sigma = bv}{id = c\ e \to c\ id = bv}\ (\text{E-CBA}) \qquad \frac{[\![e]\!]\sigma = bv \quad \sigma' = \sigma[id \mapsto bv] \quad \beta' = \text{activate}(\beta, (\sigma, \sigma')) \quad u'_{nb} = \text{trigger}(u_{nb}, (\sigma, \sigma'))}{id = e, \sigma, \beta, u_{nb}, u_\alpha \to \texttt{skip}, \sigma', \beta', u'_{nb}, \text{sched}(u_\alpha, \alpha, (\sigma, \sigma'), t)}\ (\text{E-BA})$$

$$\text{activate}(\beta, ev) = \{pid \mapsto cond \mid pid \in \text{keys}(\beta) \wedge cond = \text{onEvent}(\beta(pid), ev)\}$$

$$\text{trigger}(u_{nb}, ev) = \begin{cases} \epsilon & u_{nb} = \epsilon \\ (id, bv, \text{onEvent}(cond, ev)) +\!\!+ \text{trigger}(u'_{nb}, ev) & u_{nb} = (id, bv, cond) +\!\!+ u'_{nb} \end{cases}$$

$$\frac{upd = \text{genUpdEv}((id, e, \bot), \sigma, t)}{id \Leftarrow e, u_{nb} \to \texttt{skip}, u_{nb} +\!\!+ upd}\ (\text{E-NB}) \qquad \frac{upd = \text{genUpdEv}((id, e, c), \sigma, t)}{id \Leftarrow c\ e, u_{nb} \to \texttt{skip}, u_{nb} +\!\!+ upd}\ (\text{E-CNB})$$

$$\frac{s' = \text{peekActive}(u_{nb}) \quad s' \neq \texttt{skip}}{s, u_{nb} \xrightarrow{nb} (s \parallel s'), \text{delActive}(u_{nb})}\ (\text{Step-NB})$$

$$\text{genUpdEv}((id, e, c), \sigma, t) = \begin{cases} (id, [\![e]\!]\sigma, \checkmark) & c = \bot \\ (id, [\![e]\!]\sigma, cond) & c, \sigma, t \Downarrow_c cond, \sigma, t \end{cases}$$

$$\text{peekActive}(u_{nb}) = \begin{cases} \texttt{skip} & u_{nb} = \epsilon \\ id = bv; \text{peekActive}(u'_{nb}) & u_{nb} = (id, bv, \checkmark) +\!\!+ u'_{nb} \\ \text{peekActive}(u'_{nb}) & u_{nb} = (id, bv, cond) +\!\!+ u'_{nb} \end{cases}$$

$$\text{delActive}(u_{nb}) = \begin{cases} \epsilon & u_{nb} = \epsilon \\ \text{delActive}(u'_{nb}) & u_{nb} = (id, bv, \checkmark) +\!\!+ u'_{nb} \\ (id, bv, cond) +\!\!+ \text{delActive}(u'_{nb}) & u_{nb} = (id, bv, cond) +\!\!+ u'_{nb} \end{cases}$$

Fig. 8. Semantic rules and functions for statements.

The semantics of initial and always blocks is to create processes executing their statements concurrently in simulation. To model this behavior in λ_V, we first desugar initial blocks (`initial` s) and always blocks (`always` s) into λ_V's process items of the form "`proc` s" and "`proc while 1 then` s", respectively. Then we compose the statements in different process items using a special statement $s_1 \parallel s_2$, called *parallel composition*, during preprocessing, and set the composed statement as the first element of a configuration, as formalized by PP-Proc in Figure 6. We will explain the semantics of parallel composition below.

Figure 8 gives semantic rules for all kinds of statements in λ_V, including control-flow statements and various assignments. As mentioned before, we omit unchanged elements in *conf* in reduction relations for simplicity. Specifically, for $s_1 \parallel s_2$, by E-Par-L and E-Par-R, λ_V will execute s_1 and s_2 concurrently. The semantics of other control-flow statements are straightforward, thus, in the rest of this section, we focus on various assignments, which are the key features differentiating Verilog from software languages.

Note that the assignment we discuss in this section incorporates intra-assignment timing controls, such as the optional "c?" in $id = c?\ e$. We address this feature because it can be difficult for developers to understand, as it delays the assignment to some time later, making it challenging to reason about, especially when combined with nonblocking assignments. In λ_V, we restrict the LHS expressions of assignments to be identifiers only, to simplifying the rules; but in Full-λ_V, the LHS expressions can also be arrays and bit accesses.

3.5.1 Blocking Assignments. The semantics of blocking assignments (λ_V syntax: id = c? e) in Verilog consists of two parts: (1) updating the value of the left-hand side (LHS) variable in a store; (2) generating store changed events that can activate processes blocked by timing controls (Section 3.4) and trigger controlled update events generated by nonblocking assignments (Section 3.5.2) and continuous assignments (Section 3.6). The former is the same as what we usually call *assignment* in software languages, and the latter is a unique feature of Verilog.

Figure 8 listed all semantic rules for blocking assignments. By E-BA, when executing a blocking assignment, store changes from σ to σ'; meanwhile, a store changed event (σ, σ') is generated. This event unblocks processes that were previously waiting on related conditions by function activate(β, ev). Specifically, the function updates the associated condition of each process in β using onEvent() and if a process has its condition satisfied ($\checkmark$), it can resume execution by E-Activate in Figure 7. Moreover, store change events may affect the execution of nonblocking assignments and continuous assignments by modifying u_{nb} and u_α, which will be discussed in Sections 3.5.2 and 3.6, respectively.

Blocking assignments have another form that is guarded by an intra-assignment timing control, with syntax $id = c\ e$. Rule E-CBA shows how an intra-assignment timing control is reduced to a normal one that guards statement on the leftmost side. The key distinction between $id = c\ e$ and $c\ id = e$ is that the expression e in the former is evaluated first before the process is blocked.

3.5.2 Nonblocking Assignments. Nonblocking assignments (λ_V syntax: $id \Leftarrow c?\ bv$) have a fundamentally different behavior from blocking assignments: by their semantics, the RHS value is not assigned to the LHS variable until the end of the time step. The concept of the "end of the time step" is not the focus of this section and is formalized in Section 3.7.

To model nonblocking assignments in λ_V, we set a queue u_{nb} of controlled update events $(id, bv, cond)$ in runtime configuration, so that we can save id (the LHS variable) and bv (the RHS value) in the queue and execute the assignment later when the time is right. Here, $cond$ matters only when the assignment is guard by a timing control (by E-CNB); for other cases, we simply put $\checkmark$ in the event (by E-NB). For convenience, we define function genUpdEv$((id, e, c), \sigma, t)$ to generate controlled update events for (id, e, c). By E-BA, when the store is changed from σ to σ' by blocking assignments, we call trigger$(u_{nb}, (\sigma, \sigma'))$ to trigger the updates of conditions in u_{nb}.

After updating u_{nb} by function trigger(), rule Step-NB comes into the play. Specifically, function peekActive(u_{nb}) extracts controlled update events, whose condition is satisfied like $(id, bv, \checkmark)$, from u_{nb} and converts them to sequential blocking assignments like $id = bv; \ldots$. Finally, the resulting assignments are handed over to a new process for execution, and delActive(u_{nb}) is used to deleted controlled update events that have been handled by peekActive(u_{nb}). In Step-NB, we use reduction relation $\xrightarrow{nb}$ instead of $\rightarrow$ on configurations to ensure that the assignments are not executed until the end of the time step, and we will further explain $\xrightarrow{nb}$ in Section 3.7.

Note that in function peekActive(u_{nb}), the controlled update events are converted to assignments following the order in which they were added to u_{nb}. This order is essential as Verilog specification says that "*the order of the execution of distinct nonblocking assignments to a given variable shall be preserved.*" [ver 2006]. If the order cannot be preserved, we may fail to ensure the determinism about the result of multiple nonblocking assignments to the same variable as mentioned in Table 1. For instance, executing code `a <= #4 0; a <= #4 1;` should result in a having a value of `1` after four time steps as per the specification. If peekActive(u_{nb}) transformed the update events into parallel assignments, e.g., `a = 0 || a = 1`, non-determinism in scheduling could cause `a` to have a final value of `0` or `1`. In λ_V, queue u_{nb} and rule Step-NB preserve the order of controlled update events, allowing us to deterministically resolve the final value of `a` as `1`.

$$\frac{\begin{array}{c}u_\alpha = \{aid \mapsto (id, bv, \checkmark)\} \cup u'_\alpha \quad \sigma' = \sigma[aid \mapsto bv] \quad bv_r = \text{resolve}(id, \sigma', \kappa, \alpha) \\ \sigma'' = \sigma'[id \mapsto bv_r] \quad \beta' = \text{activate}(\beta, (\sigma, \sigma'')) \quad u'_{nb} = \text{trigger}(u_{nb}, (\sigma, \sigma''))\end{array}}{\sigma, \beta, u_{nb}, u_\alpha \to \sigma'', \beta', u'_{nb}, \text{sched}(u'_\alpha, \alpha, (\sigma, \sigma''), t)} \text{(E-ContUpdate)}$$

$$\text{sched}(u_\alpha, \alpha, (\sigma, \sigma'), t) = \text{merge}(u_\alpha, \text{signal}(\alpha, (\sigma, \sigma'), t))$$

$$\text{merge}(u_\alpha, u'_\alpha) = \{aid \mapsto (id, bv, cond) \mid u'_\alpha(aid) = (id, bv, cond) \vee (aid \notin \text{keys}(u'_\alpha) \wedge u_\alpha(aid) = (id, bv, cond))\}$$

$$\text{signal}(\alpha, (\sigma, \sigma'), t) = \{aid \mapsto \text{genUpdEv}((id, e, c), \sigma', t) \mid \alpha(aid) = (id, e, c) \wedge [\![e]\!]\sigma \neq [\![e]\!]\sigma'\}$$

$$\text{drVals}(id, \sigma, \alpha) = \{\sigma(aid) \mid \alpha(aid) = (id, \_, \_) \wedge aid \in \text{keys}(\sigma)\}$$

Fig. 9. Semantic rules and functions for continuous assignments.

3.6 Continuous Assignments

In Verilog, continuous assignments are used to model combinational logic, and in λ_V, they are represented by assign items of the form `assign` $c_d?$ id `=` e as defined in Figure 5. The semantics of continuous assignments is that when the value of e changes, the new value will be used to update id at the current time step. If a delay control c_d is given, then the update will be delayed, and triggered later as time step advances (discussed in Section 3.7).

To model continuous assignment in λ_V, we set two maps α and u_α in runtime configuration to record the `assign`-related information. Specifically, α maps each assign item (identified by aid) to a triple (id, e, c), where id and e stand for the LHS net and the RHS expression of the assignment, and c is the delay control. Accordingly, u_α maps an assign item to a controlled update event $(id, bv, cond)$, where id is the LHS net of the assignment, and bv is the new value (evaluated from e in α), and $cond$ matters only when the delay control c is not $\perp$ (otherwise, we simply put $\checkmark$).

PP-CA and PP-CCA in Figure 6 initialize α by assigning a unique identifier to each assign item and record its information in α. To maintain u_α, we define a function $\text{sched}(u_\alpha, \alpha, (\sigma, \sigma'), t)$ (in Figure 9) to generate controlled update events from assign items α whose LHS nets need updating as RHS values have been changed ($[\![e]\!]\sigma \neq [\![e]\!]\sigma'$), and then merge the events into u_α by function merge(). sched() is called every time the store σ changes, e.g., by E-BA in Figure 8.

E-ContUpdate in Figure 9 defines how to process a controlled update event $(id, bv, \checkmark)$ of aid in u_α. This rule looks a bit complicated because it has to deal with multiple drivers problem introduced in Section 3.2. Specially, for the update event $(id, bv, \checkmark)$, we first set bv for aid in store σ to maintain RHS value of each assign item (associated with aid) in the store. Then we call function $\text{resolve}(id, \sigma', \kappa, \alpha)$ to compute the new value for id, i.e., bv_r. resolve() collects the RHS values of all assign items in σ', whose LHS net is id, and resolves possible conflicts among these values, based on the resolution function specified by the net kind $\kappa(id)$ and function $\text{drVals}(id, \sigma, \alpha)$ that collects all driving values of id. The formal definition of resolve() is omitted, and please refer to Verilog specification and our artifact for details.

Note that unlike nonblocking assignments, we use a map (u_α) instead of a queue store the update events to be handled. This is to avoid the inertial delay problem mentioned in Table 1. Verilog semantics specify an inertial delay model for continuous assignments, as stated in [Gordon 1995]. If a previously-generated update event from a continuous assignment has not been executed due to a delay, and a new update event for the same net is generated, then the previously-generated event should be overwritten.

3.7 Scheduling Semantics

The scheduling semantics defines how a Verilog program is simulated, and each simulation cycle can be divided into fours phases executed in order (as introduced in Section 2.2), which are modeled by the rules in Figure 10:

$$\frac{conf \rightarrow conf'}{conf \xrightarrow{V} conf'}\ (\textsc{Sched-Norm}) \qquad \frac{conf \nrightarrow \quad conf \xrightarrow{0} conf'}{conf \xrightarrow{V} conf'}\ (\textsc{Sched-Zero}) \qquad \frac{conf \nrightarrow \quad conf \xrightarrow{0} \quad conf \xrightarrow{nb} conf'}{conf \xrightarrow{V} conf'}\ (\textsc{Sched-NB})$$

$$\frac{conf \nrightarrow \quad conf \xrightarrow{0} \quad conf \xrightarrow{nb} \quad conf \xrightarrow{1} conf'}{conf \xrightarrow{V} conf'}\ (\textsc{Sched-One})$$

$$\frac{\begin{array}{c}t' = t + \triangle \quad \beta' = \text{activate}(\beta, t') \quad u'_{nb} = \text{trigger}(u_{nb}, t') \\ u'_\alpha = \text{trigger}(u_\alpha, t') \quad (\beta, u_{nb}, u_\alpha, t) \neq (\beta', u'_{nb}, u'_\alpha, t')\end{array}}{\beta, u_{nb}, u_\alpha, t \xrightarrow{\triangle} \beta', u'_{nb}, u'_\alpha, t'}\ (\textsc{Step-Time})$$

$$\text{trigger}(u_\alpha, ev) = \{aid \mapsto (id, bv, \text{onEvent}(cond, ev)) \mid u_\alpha(aid) = (id, bv, cond)\}$$

Fig. 10. Schedule semantics and time advancement.

(1) Executing concurrent processes and continuous assignments (Sched-Norm);
(2) Unblocking zero-delayed processes and activating zero-delayed events (Sched-Zero);
(3) Executing nonblocking assignments (Sched-NB);
(4) Advancing time steps to unblock delayed processes and activate delayed events (Sched-One).

Now we can define the execution semantics of λ_V as reduction relation $\xrightarrow{V}$, which is derived from reduction relations $\rightarrow$, $\xrightarrow{\triangle}$ ($\triangle$ denotes number of advancing time steps), and $\xrightarrow{nb}$. We use $conf \nrightarrow$, $conf \xrightarrow{\triangle}$, and $conf \xrightarrow{nb}$ to denote that no reduction $\rightarrow$, $\xrightarrow{\triangle}$, and $\xrightarrow{nb}$ can be applied on $conf$.

Reduction $\rightarrow$ has been defined throughout in Section 3. Reductions $\xrightarrow{0}$ and $\xrightarrow{1}$ are defined by Step-Time, which triggers update events with delay controls in u_{nb} and u_α via functions $\text{trigger}(u_{nb}, t')$ and $\text{trigger}(u_\alpha, t')$. This behavior is the same as how blocking assignments trigger event controls as explained in Section 3.5.1, so we do not repeat it here. Specially, $\xrightarrow{0}$ handles zero-delayed processes and events. The semantics of $\xrightarrow{nb}$ (for phase (3)) have been defined in Step-NB (Figure 8), and by Sched-NB, it is executed only when no reductions for phases (1) and (2) are applicable. By rules in Figure 10, phase (i) is executed after phase (i-1), except that phase (2)–(4) may unblock some processes, then the scheduling returns to phase (1) to execute the unblocked processes.

Frozen Simulation. According to the scheduling semantics, time step in Verilog advances by one only when no processes can progress any further. Hence, if a process enters a dead loop, time step will not advance, effectively freezing the simulation. Therefore, it is incorrect to write code such as `always c = a + b` to implement an adder. Unfortunately, we still found this error in the test suite from *Icarus Verilog*.

Real-world Simulator Implementations. Although the Verilog specification precisely defines the scheduling semantics, some simulator implementations make their own assumptions about scheduling, such as using non-preemptive scheduling to address unusual behaviors in Verilog programs under specified semantics. Unfortunately, this can cause the same Verilog program to exhibit different behaviors across different simulators, or even lead semantically equivalent programs to behave differently on the same simulator, as mentioned in Table 1. λ_V faithfully follows Verilog's specification on scheduling to avoid such issues, which are further discussed in Section 5.1.

4 TOTALITY AND CONFORMANCE OF λ_V

The practical usefulness of λ_V is built upon two properties that are highlighted in previous research work of core languages [Guha et al. 2010; Politz et al. 2013]:

- *Totality*: Desugaring converts all Verilog source programs into their corresponding λ_V.
- *Conformance*: Desugared programs yield the same results as the original Verilog source.

Totality demonstrates that λ_V can work with real-world Verilog programs. Conformance ensures that λ_V is reliable and trustworthy. However, the conformance property cannot be proven due to the absence of a complete formal specification for Verilog's behavior. To tackle both properties, we follow the testing strategy adopted in existing work for Python [Politz et al. 2013] and JavaScript [Guha et al. 2010] as depicted in Figure 1. This method treats real-world simulators as the closest approximation of the missing Verilog's complete formal specification. By implementing λ_V and comparing its results with those of simulators on sufficient number of Verilog programs, we can test λ_V's totality and conformance. Below, we discuss the key aspects of implementing and testing respectively.

4.1 Implementation

We developed λ_V in Java, which comprises about 27,000 lines of code (excluding comments). Our project mainly consists of three parts: a general-purpose Verilog front end that converts Verilog to an IR (about 15,000 lines of code), a desugar function that further transforms the IR to λ_V (about 2,000 lines of code), and the λ_V part (about 8,000 lines of code).

Front End. Verilog supports meta-level language features, such as `generate` and `parameter`, to elaborate instantiations of modules. To represent such instantiated code, we require an elaborate process. This involves following the specification to "bind modules to their instances, build the model hierarchy, compute parameter values, resolve hierarchical names, process generate constructs, and establish net connectivity". To our knowledge, these meta-features of Verilog are overlooked by other semantic works in Table 2, despite their widespread use in the real world; thus, we support them to enhance the totality of λ_V.

Consequently, the front end is organized into two phases. The first phase involves parsing Verilog programs into abstract syntax trees (ASTs) based on ANTLR [Parr 2013]. Subsequently, in the second phase, these ASTs are transformed into their elaborated versions, represented by a specialized IR designed by us.

Desugaring. The desugar function involves desugaring Verilog's various features on the IR, and converting IR into the representation of λ_V. Below, we highlight several important operations in desugaring.

- We make explicit any implicit type conversions in context-determined expressions, which addresses the pitfall discussed in Section 3.3.
- We unify the representation of bit vectors. Verilog allows for bit vectors to be represented in both big-endian and little-endian forms, and negative indices are also permitted. We transform all bit vectors into little-endian representation and shift their ranges to ensure their indices start from 0.
- We process all forms of data declarations, which can be a complicated process due to the flexibility of Verilog's data declaration rules. In Verilog, data identifiers can be declared separately from their other information, such as data types, or used without prior declaration.
- We process the alias semantics under `inout` port connections by replacing all occurrences of aliased nets with a fresh uniformed name. To our knowledge, no existing semantics work handles `inout` port connections.

λ_V. The λ_V part includes a parser based on ANTLR [Parr 2013] for parsing λ_V's textual representation, data structures representing λ_V's syntax, and the faithful implementation of semantic rules, based on which our interpreter is implemented.

To make our interpreter applicable to real-world programs, we have modeled numerous system tasks and functions that are frequently encountered in real-world programs, such as `$monitor`

Table 3. The distribution of features in the first test suite. Since a test case may contain multiple features, we count them separately in each feature category. Feature abbreviations can be found in Table 1.

	BV	N	CE	TC	BA	NBA	PCA	AFC	CA	SCH	CONN	LT	STF	GATE	Total
# of Test Cases	824	342	730	542	653	83	44	24	293	539	303	105	824	28	824

(enabling variable change tracing), `$display` (permitting the printing of formatted messages), and `$random` (enabling random data generation for testing), and so on.

4.2 Testing

To test the totality and conformance of λ_V, we employ two distinct test suites, each evaluating λ_V from different aspects:

- The first test suite comprises the test cases that comprehensively cover various language features. They are sourced from *Icarus Verilog* [Williams 2023], one of the two most widely used open-source simulators for Verilog. Compared to the other simulator, i.e., *Verilator* [Snyder 2023a], *Icarus Verilog* offers much better support on resolving language features (*Verilator* fails to pass 100+ test cases of *Icarus Verilog* in our experiment, so we will not describe *Verilator* in the rest of this section). We exclude test cases that contained specific compiler directives and the features not included in the standard specification of Verilog [ver 2006], such as *Icarus Verilog*'s specific extensions, non-standard system functions, as well as the rare features that λ_V does not support, as explained in Section 1. In total, we obtained 824 test cases, each residing in a separate file and collectively comprising about 28,000 lines of code.
- The second test suite is derived from real-world programs that tightly combine language features. It includes a suite of unit tests (including the one using the largest test data) from OpenPiton [Balkind et al. 2016], an open-source processor, and additional test cases primarily sourced from Altera Corporation, encompassing a range of designs such as an rv32i CPU, with code sizes varying from hundreds to thousands of lines. Note that not all unit tests from OpenPiton are included in our suite: we have excluded tests that utilize configurations specific to FPGA platforms and those that report configuration errors due to missing files.

Totality. Table 3 shows the distribution of all the representative features of Verilog, as summarized in Table 1, within our first test suite. It is important to note that since a test case may utilize multiple features, the sum of cases for each feature does not necessarily equal the total number of test cases. As explained in Section 1, no existing semantic work has been able to handle these features as comprehensively as λ_V. In our experiment, λ_V's implementation successfully converted all those 824 test cases written in Verilog to λ_V, demonstrating its good totality for desugaring all representative features of Verilog.

In the second test suite, the tests from OpenPiton were not directly compatible with our frontend due to the presence of compiler directives and specific dialects used by various commercial simulators. To address this incompatibility, we utilized the tool *vppreproc* [Snyder 2023b] to process any compiler directives and then manually replaced those specific dialects with their standard Verilog equivalents. After completing this process, our implementation successfully converted all those test cases of OpenPiton to λ_V; moreover, the other real-world programs in our second test suite mentioned earlier were also converted successfully, showcasing λ_V's good totality in handling real-world programs.

Conformance. During our testing using the first test suite, we observed cases in which λ_V failed to pass due to semantic bugs in *Icarus Verilog*, indicating a failure on the part of *Icarus Verilog* to adhere to the standard specification of Verilog. We will examine this issue in Section 5.1. Additionally,

some failed cases were found to be caused by ambiguities in the Verilog specification, leading to inconsistent outputs across different simulators. This will be discussed in Section 5.2.

In addition to the above cases, λ_V only fails to pass nine cases out of the 824 cases presented in Table 3. However, upon further investigation of the root causes for the failed test cases, we discovered that they do not necessarily indicate non-conformance of λ_V with Verilog. After careful analysis, we identified the root causes of those failed test cases, which can be categorized as follows:

- Seven test cases were written erroneously by their developers. For instance, upon inspecting one of the tests, we discovered that the variable `r` should hold bit values of either `0` or `x`, but the developer checked it with the assertion "`r==1`", mistakenly using "==" instead of "===". This changes the semantics under which `0` would be considered a wrong result for `r`. (Note that the difference in semantics between "===" and "==" can be complicated when combined with the value `x` in Verilog, so let us ignore it and focus on its effect instead). *Icarus Verilog* can pass this test as it always outputs `x` for `r` due to its limited scheduling strategy as introduced in Section 5.1.2, while λ_V faithfully follows the specification's scheduling, resulting in `0` being produced for `r` under some of its scheduling, causing the test to fail.
- Two test cases do not pass λ_V's type checker. The Verilog specification prescribes issuing warnings when certain implicit conversions or coercions occur. λ_V implements a more rigorous type checking process during compilation, leading to the rejection of programs that use implicit port kind coercion and dissimilar net types resolution. Popular Verilog source code linting tools typically report similar warnings to be fixed, and thus we argue that it is advisable to eliminate such warnings during development to avoid unexpected behavior.

It's worth noting that *Icarus Verilog* actually encountered three failures out of the 824 cases due to a known bug resulting from an incorrect implementation of procedural continuous assignment. In contrast, λ_V successfully passed these cases.

Based on the explanation outlined above, it is reasonable to conclude that λ_V conforms with Verilog in all the representative features we have tested.

Additionally, λ_V successfully passed all the test cases from our second test suite involving real-world programs. For example, λ_V completed one unit test with the largest test data from OpenPiton in approximately one hour on a laptop, generating several megabytes of output over approximately 2,000,000 clock cycles.

5 USEFULNESS OF λ_V

The semantics and implementation of λ_V offer various utilities in a range of contexts. Specifically, in this section, we highlight how λ_V uncovers real-world semantic bugs that are often perplexing to Verilog programmers and are even overlooked by popular Verilog simulators (Section 5.1). Additionally, we show how λ_V exposes ambiguities in Verilog's standard specification (Section 5.2), and discuss how λ_V and its interpreter can facilitate other applications, such as exploring a circuit's state space and building a concolic execution tool for Verilog (Section 5.3).

5.1 Detect Real-world Semantic Bugs

Thanks to the more accurate modeling of Verilog's semantics by λ_V, along with its properties of totality and conformance, we can rely on λ_V to detect semantic bugs in Verilog by comparing its results with those of other real-world simulators. In other words, if a test passes on other simulators but not on λ_V's interpreter, it is possible that the compared simulator has a semantic issue, except in cases as explained in Sections 1 and 4.2.

As a result, λ_V identified real-world semantic bugs that were even erroneously interpreted by the two most popular open-source Verilog simulators, *Icarus Verilog* [Williams 2023] and

Verilator [Snyder 2023a] (as *Icarus Verilog* has stricter semantic treatment compared to *Verilator*, we will only use *Icarus Verilog* for illustration for the remainder of the paper). These bugs fall into three categories including context-determined expressions, hidden data races, and lost of newest results. Due to limited space, we illustrate the first two kinds of semantic bugs in this section, and we will describe the remaining bugs in the artifact.

5.1.1 Context-determined Expressions. Evaluating context-determined expressions is a persistent issue that continuously disturbs Verilog programmers and developers of language tools as discussed in Section 3.3. Let us consider a real-world example about using bit shifts. This case concerns the use of system function `$signed()` in context-determined expressions, which can lead to unexpected implicit conversions and cause confusing behavior by shift operation. Specifically, programmers often write the following code to express the intent of applying an arithmetic or logical shift right operator to bit vector `val`, depending on the `ctl` signal:

```
ctl ? $signed(val)>>>offset : val>>offset
```

Let us assume that `ctl`, `val`, and `offset` are assigned the values `1'b1`, `4'b1010`, and `4'b0001`, respectively. In this case, `ctl` evaluates to `true`, and the expression `$signed(val)>>>offset` will be evaluated. Intuitively, we might assume that `val` is of a signed type, and therefore the one-bit right shift operation should retain the high bit, which is `1`. Consequently, the expression should evaluate to `4'b1101`. However, this is incorrect, and the correct result should be `4'b0101`, where the high bit is `0` instead of `1`. Such incorrect evaluation has persisted in the old versions of *Icarus Verilog* for an extensive period; moreover, we can see a frequent recurrence of a similar question on website *Stack Overflow*, which continues to confound Verilog programmers even today.

This is because the influence of context type in this case has been ignored. As per the specification of Verilog, the two subexpressions, i.e., `$signed(val)>>>offset` with a signed 4-bit type and `val>>offset` with an unsigned 4-bit type, have an impact on each other's context type. Therefore, the final context type is their least common type, which is an unsigned 4-bit type. Consequently, this context type is then applied as the context type of `$signed(val)`, which converts `$signed(val)` to an unsigned value. Thanks to λ_V's accurate modeling of semantics, it is able to identify this confusing issue and produce correct result for it.

5.1.2 Hidden Data-races. One problem that is often overlooked in Verilog simulators is data races, which occur when multiple processes access the same variable without proper synchronization. In this section, we illustrate a case where *Icarus Verilog* hides a potential data race.

```
1 always @(*) c1 = a + b;
2 assign c2 = a + b;
3 initial begin
4   a = 1; b = 2; #0;
5   if (c1 == c2) $display("same");
6   else $display("different");
7 end
```

Consider the example program on the side, which models the same combinational logic using both the `always @(*)` statement (line 1) and the continuous assignment (the assign statement on line 2), where `a`, `b`, and `c1` are 32-bit variables, while `c2` is a 32-bit net. Verilog programmers are often taught that the effects of the first two lines of code are equivalent, and thus `c1` should equal `c2`, and "`same`" will be displayed. However, this is not always the case, as both "`same`" and "`different`" outputs can be displayed in this example. This occurs because the process created by the `always` block has data races with the process created by the `initial` block, which could lead the program to print `"different"`. Below, we give one such schedule. For simplicity, we denote the two processes created by the `always` block and the `initial` block as P_1 and P_2, respectively.

(1) Execute `@(*)` in P_1 so that P_1 is blocked and waits for any changes in the values of its listening variables, `a` and `b`. (wildcard `*` means it listens to changes in either `a` or `b`).

(2) Execute `a = 1` in P_2. The event that `a`'s value has been changed is then sent to other listening variables, such as `*` in P_1.
(3) As `a`'s value has been changed, `a + b` in P_1 can be evaluated, where `a` has the value of `1` and `b` has the default value of `x`, resulting in a value of `x` that is then assigned to `c1` in P_1.
(4) Execute `b = 2` in P_2, and then the zero delay (`#0`). Executing `#0` will block the current process P_2, allowing the schedule to execute other processes and continuous assignments.
(5) Execute `@(*)` in P_1 again to listen to changes in `a` and `b` (`always` block is actually a loop). However, it misses the change in `b`'s value in the last step. Hence, the value of `c1` remains `x`.
(6) Execute the continuous assignment and `c2` is assigned the value of `a + b = 1 + 2 = 3`.
(7) Execute the previously blocked P_2 to compare the values of `c1` and `c2`. As `x` does not equal `3`, "`different`" is displayed using the `$display` system task.

According to the specification of Verilog, the code above should be able to print "`same`" or "`different`" given different schedules, and we can utilize λ_V to generate schedules to output either result; however, *Icarus Verilog* can only output "`same`" in this case, which violates the language specification and hides the potential data races. This is because *Icarus Verilog* always executes the `@(*)` of processes first and only supports non-preemptive scheduler. In the above example, after compulsively executing `@(*)` of P_1, *Icarus Verilog* must execute `a = 1` and `b = 2` incessantly in P_2 as it is not allowed to be preempted until it releases the schedule proactively (for example, by executing zero delay to block itself in our example). As a result, both `c1` and `c2` equal `3` and "`same`" will always be displayed. *Icarus Verilog* is designed to guide executions by allowing only a subset of scheduling, intentionally hiding potential data races from programmers on the basis that synthesized circuits do not contain data races. However, because existing simulators cannot completely hide data races, we argue that *Icarus Verilog*'s limited scheduling for hiding data races may introduce issues during simulation that can confuse programmers and complicate debugging.

For the example above, *Icarus Verilog* forces execution of `@(*)` first to always display "`same`" (indicating no data race). But when we reorder the initial process and always block, and make `@(*)` the one that explicitly enumerates the listening variables' posedge/negedge, "`different`" will be displayed (indicating the presence of data race). However, according to the specification and intuition, programmers expect that the code in each orderings should have a chance to display "`different`"; nevertheless, *Icarus Verilog* outputs unexpected results by adding constraints for scheduling, violating the specification and confusing programmers. This also adds to the burden of debugging, as programmers may be unaware of whether unexpected results are due to data race hiding. Therefore, we advocate scheduling as per the language specification to expose data races. If possible, we propose taking it a step further by advocating for Verilog to provide language support that unifies the semantics of always blocks for combinational logic, and continuous assignments, for addressing any potential data races. This will result in consistent behavior across various simulators and simplify the process of reasoning about code behavior for Verilog programmers.

5.2 Expose Ambiguities in Specification

Compared to software languages, the specification of Verilog contains more unclear definitions and descriptions. With the help of λ_V, we have been able to identify seven cases with certainty where inconsistent results are produced by different simulators due to the ambiguities in specification. These cases cover four types of scenarios including timing-control assignments, procedural continuous assignments, unmatched port connections, and even the syntax regarding named events. Due to limited space, we explain one simple example about timing-control assignments (the code is shown below), and the remaining cases will be described in the artifact.

```
wire [1:0] a;
assign #1 a = 2'b10;
assign a = 2'b01;
initial begin
  #0 $display("%d", a);
end
```

The code shows that net a is assigned by two drivers, one with a delay control (#1) and one without. Due to the concurrent scheduling of Verilog, the two `assign` statements and the `$display` statement are executed in parallel. According to the specification, "*The default initialization value for a net shall be the value 'z'. Nets with drivers shall assume the output value of their drivers*". However, the specification does not clarify what should happen if the driver's output value is not available during initialization. In our case, during time step zero, the first `assign` statement will not drive the value `2'b10` to net a until the next time step due to the delay of #1. As a result, λ_V assumes that net a will be assigned the value 'z'. Further because the second `assign` statement assigns the value `2'b01` to net a in parallel, a multiple-driver problem occurs, leading to value 1 being displayed as explained in Section 3.2. However, upon inspection of *Icarus Verilog*'s implementation, we found that *Icarus Verilog* assumes that the value x will be assigned to net a; thus, the final value displayed is 'x'. The inconsistent results are due to the ambiguity in the specification.

After reviewing our experience, we have realized that the Verilog specification appears to be less clear than that of software languages. Our testing process may only uncover a certain portion of these ambiguities and we highly recommend a thorough refinement of Verilog's standard specification for disambiguations.

5.3 Facilitate Other Applications

Developing *proofs* and *tools* for a core language that captures the essence of language P is much easier than addressing the full details of P itself [Krishnamurthi 2015]. Just as the core languages for JavaScript [Guha et al. 2010] and Python [Politz et al. 2013], λ_V is such an essence-capturing language for Verilog that possesses the desirable properties of totality and conformance, as demonstrated in Section 4.1. As a result, the advantage of using λ_V to facilitate proof development is easy to understand: many of the complicated features of Verilog's semantics have been desugared and flattened, possibly making *proofs* significantly shorter for λ_V than for Verilog; in this section, we briefly discuss how we utilize λ_V and its interpreter to develop potential *tools* more easily.

5.3.1 State-Space Explorer. Using simulators like *Icarus Verilog*, Verilog programs are scheduled along a specific execution path for stability. However, since Verilog's concurrency control is feature-rich, selecting a specific (or subset of) scheduling in simulation may obscure potential issues. For instance, what if a program passes in *Icarus Verilog* but not in another simulator? (this issue may remain undetected if only *Icarus Verilog* is used). To address this issue, one possible solution is to rely on a Verilog state-space explorer that can traverse all possible execution states of all schedulings. With such explorer, it can be determined whether the program can pass only when following *Icarus Verilog*'s limited scheduling, exposing a fault in *Icarus Verilog*, or if it can pass through all schedules, verifying the correctness of *Icarus Verilog* (and exposing the fault of the other simulator).

λ_V offers a scheduling with monotonicity [Vafeiadis et al. 2015] and fairness by accurately adhering to the specification of Verilog; moreover, we have encoded λ_V's interpreter with auxiliary APIs that enable the convenient construction of initial states and computation of all possible next states allowed by λ_V's semantic rules for any input Verilog program (the term "state" refers to the configuration defined in Section 3). With this facilitation, we were able to effortlessly construct a Verilog state-space explorer by appending just over 100 lines of code to the interpreter of λ_V. This explorer aided in identifying data race issues as described in Section 5.1.2, and enabled us to promptly confirm concurrency-related issues across various simulators. We use the state-space explorer as necessary, and based on our experience, it can take anywhere from a few seconds to several minutes to complete. For instance, for a test case with 40+ lines of code containing around 50,000 schedules, the state-space explorer took 11 minutes to run.

5.3.2 Concolic-Execution Tool. Concolic execution [Godefroid et al. 2005] is a useful testing approach for guiding executions to cover more software code. Similarly, there is related work for hardware [Ahmed et al. 2018; Meng et al. 2022]. Its basic working mechanism is as follows: given an input, Verilog programs are executed and their execution traces are extracted and collected to generate constraints; symbolic execution traces are then created, followed by selecting a constraint for flipping to explore other branches. Subsequently, the flipped constraint is solved to generate new input for guiding new executions.

After developing our concolic-execution tool, we discovered that creating such a tool on top of λ_V and its interpreter required significantly less effort than building on top of traditional simulators like *Icarus Verilog*, as noted in previous work [Ahmed et al. 2018]. The main reasons are as follows.

Extracting execution traces from simulators such as *Icarus Verilog* and *Verilator* can be a complicated task, as users are unable to directly obtain traces through simulator interfaces. As a result, previous work [Ahmed et al. 2018] has required the insertion of monitoring code into the program to capture execution information, which can only be extracted once the simulation is complete. In contrast, the concise semantics of λ_V and its extensible interpreter implementation have enabled us to obtain real-time execution information with minimal effort. By adding just a few dozen lines of code to the interpreter, we can capture information such as the value of a variable at a particular time or the statement being executed at that moment.

Generating constraints with constraint solvers like Z3 [de Moura and Bjørner 2008] often requires strict adherence to expression types. For example, Z3's arithmetic operations require bit vectors of equal width as operands. Verilog, as a weakly-typed language, necessitates manual handling of various type conversions when generating constraints. In contrast, all primitive functions in λ_V are strongly-typed and feature automatic type conversion (see Section 3.3), making the process of generating constraints from λ_V statements significantly easier. Moreover, λ_V has inlined all modules and functions, reducing the need for instantiating modules and invoking functions when generating constraints for λ_V programs, which further simplifies the process of constraint generation.

We enlisted the help of a senior undergraduate student to implement a concolic-execution tool using λ_V's interpreter by referencing λ_V's semantics (he was not familiar with Verilog previously). As of the time of writing, the tool is functionally equivalent to similar work [Ahmed et al. 2018] and required only approximately 100 working hours to complete. Furthermore, the tool is scalable to the largest Verilog design (about 8,000 lines of code) in our real-world program testbench (the largest benchmark program in the LLHD paper [Schuiki et al. 2020] contains about 4,000 lines of code), which takes approximately 4 minutes to run a cycle concolically on the student's laptop, and the time increases linearly with the number of cycles. Once again, this result demonstrates the potential value of λ_V and reinforces Krishnamurthi [2015]'s perspective that research groups with limited resources can leverage core languages (like λ_{JS} [Guha et al. 2010], λ_π [Politz et al. 2013], and our λ_V) to apply their tools to real programs, enhancing the utility of their research.

6 RELATED WORK

Operational Semantics for Verilog. Our focus is on discussing operational semantics for Verilog, similar to our own, as summarized in Table 2. Other styles of semantics for Verilog are discussed in the next part.

The earliest work in this area, [Gordon 1995], discusses the semantic challenges of commonly-used Verilog features, but only provides informal descriptions. Other early works, such as [Fiskio-Lasseter and Sabry 1999; He and Xu 2000; He and Zhu 2000], do not cover several key features of Verilog, such as continuous assignments and nonblocking assignments.

Later work, [Dimitrov 2001], covers the key features but falls into almost all of the pitfalls discussed in this paper, as depicted in Table 1. Furthermore, their approach to modeling Verilog

is not scalable to support missing Verilog features (e.g., advanced flow control), in the sense that supporting those features would require rewriting most of their semantic rules.

The most complete work on Verilog semantics is [Meredith et al. 2010], which is based on the K framework [Meseguer and Rosu 2007] and covers many representative features discussed in this paper. However, it fails to handle several pitfalls. Firstly, they do not model the x and z bits in Verilog. Secondly, they do not fully capture the essential differences between nets and variables, resulting in their semantics being unable to handle the problem of multiple drivers. Thirdly, they do not model repeat event controls, procedural continuous assignments, and advanced flow controls, which are hard to extend based on their semantic core. Finally, they fail to model the alias semantics under the inout port of modules.

Recent years, Lööw and Myreen [2019] propose Verilog's formal semantics as to build a verified translator from HOL functions to Verilog programs. However, their semantics is designed to only model a synthesizable subset of Verilog that they focus on. In contrast, λ_V aims to provide a complete formal semantics for Verilog. It is worth mentioning that it is relatively straightforward to tailor λ_V to obtain the subset of Verilog used by Lööw and Myreen [2019], which demonstrates λ_V's potential in verification scenarios. In addition, λ_V can assist Verilog tool developers in identifying potential pitfalls comprehensively in advance and reduce problems when the scope of their tools is extended. In fact, Lööw [2022] mentions that they have encountered a concurrency problem in Verilog's semantics that has temporarily sidetracked their ongoing Verilog semantics formalization attempt. The concurrency problem has been addressed and discussed in our paper, and we expect λ_V to provide help to their work in this regard. Herklotz et al. [2021]'s later work on Coq extends the semantics in Lööw and Myreen [2019], but its scope is still limited to the synthesizable Verilog, and thus we do not delve into further discussion.

It is important to note that none of the above works is a core language like λ_V, and thus, they may lack certain advantages of an essence-capturing language, as discussed in Section 5.

Other Styles of Semantics. Apart from operational semantics, various other styles of semantics have been explored for Verilog, including denotational semantics [Huibiao and Jifeng 2000], trace semantics [Gordon 2002], and algebraic semantics [Zhu et al. 2008]. However, these works are incomplete and fail to cover many key features of Verilog.

This does not mean other styles of semantics are not the right choice for modeling Verilog semantics, as no definitive conclusion has been reached regarding this topic. However, we have opted for the operational approach. We found that operational semantics offers two advantages in the context of Verilog. Firstly, it provides an intuitive representation of Verilog's stateful transition system, making it easier for readers to grasp the language's behavior. In our experience, papers employing operational semantics for Verilog tend to be more accessible, and we have observed many researchers adopting this approach to formally describe features and constructs of software languages. Additionally, operational semantics provides a simplified model that aligns well with the implementation of simulators adhering to the event-driven style scheduling semantics specified in Verilog. This can help explain perplexing behaviors encountered when using those Verilog simulators. We guess that this preference for operational semantics in previous work on core languages, such as λ_{JS} [Guha et al. 2010] and λ_π [Politz et al. 2013], may also be attributed to the accessibility of operational semantics.

It is worth noting that process calculus may present an option for modeling the concurrency in Verilog. However, no previous work has employed this approach to model Verilog semantics. We acknowledge that process calculus may be better suited for modeling concurrent systems that communicate through message passing, akin to real-world hardware where electronic components communicate concurrently via messages transmitted on physical wires. However, Verilog models

hardware using concurrent processes created by always blocks that communicate through shared variables. Should λ_V adopt process calculus, the desugared Verilog program might undergo significant transformation, necessitating the conversion of shared variable communication to message passing. Such a departure from the original Verilog model could potentially contradict our design intention of enabling readers to smoothly transfer their understanding from λ_V to Verilog.

Intermediate Representations. λ_V can be used as an intermediate representation (IR) for simplifying downstream tasks, but it significantly differs in design intention from existing Verilog IRs that we have already known. Generally speaking, λ_V, as a semantic work, places a higher priority on completeness, aiming to present readers with a comprehensive understanding of the pitfalls in Verilog, even if it entails introducing more constructs to achieve this goal. In contrast, other IRs designed for Verilog prioritizes tractability by sacrificing some completeness beyond their specific application scopes. The following paragraphs compare λ_V with several popular IRs that cover three typical application scenarios: synthesis, simulation, and verification.

LLHD [Schuiki et al. 2020] is a remarkable contribution that provides a multi-level IR for representing digital circuits throughout the entire design flow, e.g., synthesis, simulation, and verification. It provides a few orthogonal primitives to model the complex features of Verilog that distinguish it from software languages, such as timing controls and unique procedural statements, etc. However, when these Verilog features are jointly used, LLHD falls short in fully supporting them, whereas λ_V excels.

For instance, LLHD lacks support for the mixed use of blocking assignments and nonblocking assignments. It transforms both types of assignments into a unified static single assignment form, which obfuscates the semantic distinction between them. This transformation works well for scenarios where Verilog programs are synthesizable and the mixing of nonblocking and blocking assignments is prohibited. However, as a semantic work rather than an IR, λ_V strives to accurately differentiate between blocking and nonblocking assignments, providing readers with formal semantics to prove when these two assignment types can be safely unified. This way, language tools like LLHD can confidently ensure that, within their scope, they can sacrifice some completeness for greater tractability.

Another example relates to the support for timing controls and nonblocking assignments. LLHD offers two orthogonal primitives: timing controls through wait instructions and nonblocking assignments through static single assignments. In typical scenarios where timing controls and nonblocking assignments are used, such as `@(posedge clk) id <= e` in Verilog, LLHD can easily express them using a wait instruction followed by assignments. However, when these two features are combined in a different manner, such as `id <= @(posedge clk) e`, LLHD cannot express it according to the Verilog specification using the mechanisms provided by LLHD like sequential composition of wait instructions and assignments. In contrast, λ_V employs three non-orthogonal primitives ($c := \ldots, s := c\ s \mid id \Leftarrow c?\ e \mid \ldots$) to support all features related to timing controls and nonblocking assignments, as depicted in Figure 5. This design decision in λ_V highlights its prioritization of completeness as a semantic work, aiming to alert readers to potential pitfalls in Verilog, such as the inability to desugar $id \Leftarrow c\ e$. Developers who want to design an IR like LLHD can then choose to delete the $c?$ in $id \Leftarrow c?\ e$ if they are confident that such forms of nonblocking assignments will not occur. In doing so, the definition of λ_V statements can be simplified to $s ::= c\ s \mid id \Leftarrow e \mid \ldots$ which are precisely orthogonal, similar to LLHD.

FIRRTL [Izraelevitz et al. 2017] is another well-known IR commonly employed for synthesis and simulation, which is used by Chisel [Bachrach et al. 2012], a popular hardware construct language, and ESSENT [Beamer and Donofrio 2020], an impressive simulator optimized for speed. However,

FIRRTL is a highly simple IR with lower expressiveness compared to LLHD, so we do not discuss it further.

ACL2 [Kaufmann et al. 2013] is a language that was successfully applied to verify chips. The SV library in ACL2 provides an IR called SVEX, which is designed to represent RTL designs and can be used for simulation and verification. However, SVEX does not offer sufficient constructs to fully support Verilog features as we do. If we were to find a reference point, SVEX's expressiveness is comparable to that of LLHD, whose difference with λ_V has been explained previously.

Core Languages. The development of our Verilog semantics has been greatly influenced by related works that define semantics using core languages. Notably, prior research on JavaScript [Guha et al. 2010; Politz et al. 2012] and Python [Politz et al. 2013] has successfully defined semantics for popular programming languages using core languages. These works demonstrate the advantages of using core languages to facilitate tools and proofs and provide insights into the language.

Motivated by their experiences and findings, we designed λ_V as a core language to describe a more comprehensive semantics for Verilog and to clarify the pitfalls in its semantics. Additionally, [Krishnamurthi 2015] emphasizes the opportunities and challenges of desugaring, which guided our implementation of converting Verilog programs to λ_V.

Furthermore, [Krishnamurthi et al. 2019] surveyed the use of core languages in defining semantics for programming languages and argued that this approach has significant advantages over approaches that rely on hundreds of semantic rules for bare languages. We believe that the usefulness of λ_V provides another evidence to support this argument.

7 CONCLUSION

The rapid growth of scaled and customized hardware has made the quality of hardware increasingly important. Verilog, as the most popular hardware description language, its semantics plays a critical role in shaping the correctness of any analysis, verification, and simulation tools for Verilog. This paper presented λ_V, a core language for Verilog that captures the essence of Verilog with much fewer language structures, covering the most complete set of features to date, while addressing the pitfalls in semantics. The implementation of λ_V was comprehensively tested, and its usefulness was demonstrated through various applications, such as detecting real-world semantic bugs, exposing ambiguities in standard specifications, and facilitating tools for Verilog. While developing λ_V, we came to strongly feel the practical usefulness of having a core language for describing semantics, echoing the sentiments of previous literature on software languages such as λ_{JS} and λ_π. As a result, we will open-source λ_V, which contains about 27,000 lines of Java code (excluding comments), and expect it to be advantageous for numerous potential applications in the future.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their helpful comments. This work was supported by the Natural Science Foundation of China under Grant Nos. 62025202, 62002157 and 61932021, the Leading-edge Technology Program of Jiangsu Natural Science Foundation under Grant No. BK20202001, and the Fundamental Research Funds for the Central Universities of China under Grant Nos. 020214380102 and 020214912220. The authors would also like to thank the support from the Collaborative Innovation Center of Novel Software Technology and Industrialization, Jiangsu, China.

DATA-AVAILABILITY STATEMENT

We have made available an artifact [Chen et al. 2023a], which encompasses comprehensive documentation of all detected bugs and ambiguities not covered in this paper. Additionally, it serves as a means to reproduce the testing results discussed in Section 4. The artifact can be downloaded from

the following URL: https://doi.org/10.5281/zenodo.8320642. To reproduce the results, please refer to the instructions provided in the accompanying `README.pdf` document within the artifact package.

Furthermore, we have provided supplementary material [Chen et al. 2023b] to elaborate the syntax and semantics of the Full-λ_V. You can access this supplementary material at the following URL: https://doi.org/10.5281/zenodo.8321324.

REFERENCES

2006. IEEE Standard for Verilog Hardware Description Language. *IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001)* (2006), 1–590. https://doi.org/10.1109/IEEESTD.2006.99495

Alif Ahmed, Farimah Farahmandi, and Prabhat Mishra. 2018. Directed test generation using concolic testing on RTL models. In *2018 Design, Automation & Test in Europe Conference & Exhibition, DATE 2018, Dresden, Germany, March 19-23, 2018*, Jan Madsen and Ayse K. Coskun (Eds.). IEEE, 1538–1543. https://doi.org/10.23919/DATE.2018.8342260

Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. 2012. Chisel: Constructing Hardware in a Scala Embedded Language. In *Proceedings of the 49th Annual Design Automation Conference* (San Francisco, California) *(DAC '12)*. Association for Computing Machinery, New York, NY, USA, 1216–1225. https://doi.org/10.1145/2228360.2228584

Jonathan Balkind, Michael McKeown, Yaosheng Fu, Tri Nguyen, Yanqi Zhou, Alexey Lavrov, Mohammad Shahrad, Adi Fuchs, Samuel Payne, Xiaohua Liang, Matthew Matl, and David Wentzlaff. 2016. OpenPiton: An Open Source Manycore Research Framework. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems* (Atlanta, Georgia, USA) *(ASPLOS '16)*. ACM, New York, NY, USA, 217–232. https://doi.org/10.1145/2872362.2872414

Scott Beamer and David Donofrio. 2020. Efficiently Exploiting Low Activity Factors to Accelerate RTL Simulation. In *57th ACM/IEEE Design Automation Conference, DAC 2020, San Francisco, CA, USA, July 20-24, 2020*. IEEE, 1–6. https://doi.org/10.1109/DAC18072.2020.9218632

Qinlin Chen, Nairen Zhang, Jinpeng Wang, Tian Tan, Chang Xu, Xiaoxing Ma, and Yue Li. 2023a. *The Essence of Verilog: A Tractable and Tested Operational Semantics for Verilog (Artifact)*. https://doi.org/10.5281/zenodo.8320642

Qinlin Chen, Nairen Zhang, Jinpeng Wang, Tian Tan, Chang Xu, Xiaoxing Ma, and Yue Li. 2023b. The Essence of Verilog: A Tractable and Tested Operational Semantics for Verilog (Supplementary Material). https://doi.org/10.5281/zenodo.8321324

Leonardo Mendonça de Moura and Nikolaj S. Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings (Lecture Notes in Computer Science, Vol. 4963)*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

Jordan Dimitrov. 2001. Operational Semantics for Verilog. In *8th Asia-Pacific Software Engineering Conference (APSEC 2001), 4-7 December 2001, Macau, China*. IEEE Computer Society, 161–168. https://doi.org/10.1109/APSEC.2001.991473

John Fiskio-Lasseter and Amr Sabry. 1999. Putting Operational Techniques to the Test: A Syntactic Theory for Behavioral Verilog. *Electronic Notes in Theoretical Computer Science* 26 (1999), 34–51. https://doi.org/10.1016/S1571-0661(05)80282-8 HOOTS '99, Higher Order Operational Techniques in Semantics.

Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed Automated Random Testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation* (Chicago, IL, USA) *(PLDI '05)*. Association for Computing Machinery, New York, NY, USA, 213–223. https://doi.org/10.1145/1065010.1065036

Steve Golson and Leah Clark. 2016. Language wars in the 21st century: verilog versus vhdl–revisited. *Synopsys Users Group (SNUG)* (2016).

Michael J. C. Gordon. 1995. The Semantic Challenge of Verilog HDL. In *Proceedings, 10th Annual IEEE Symposium on Logic in Computer Science, San Diego, California, USA, June 26-29, 1995*. IEEE Computer Society, 136–145. https://doi.org/10.1109/LICS.1995.523251

Michael J. C. Gordon. 2002. Relating Event and Trace Semantics of Hardware Description Languages. *Comput. J.* 45, 1 (2002), 27–36. https://doi.org/10.1093/comjnl/45.1.27

Tomás Grimm, Djones Lettnin, and Michael Hübner. 2018. A Survey on Formal Verification Techniques for Safety-Critical Systems-on-Chip. *Electronics* 7, 6 (2018). https://doi.org/10.3390/electronics7060081

Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. 2010. The Essence of JavaScript. In *ECOOP 2010 - Object-Oriented Programming, 24th European Conference, Maribor, Slovenia, June 21-25, 2010. Proceedings (Lecture Notes in Computer Science, Vol. 6183)*, Theo D'Hondt (Ed.). Springer, 126–150. https://doi.org/10.1007/978-3-642-14107-2_7

Jifeng He and Qiwen Xu. 2000. An Operational Semantics of a Simulator Algorithm. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, PDPTA 2000, June 24-29, 2000, Las Vegas, Nevada, USA*, Hamid R. Arabnia (Ed.). CSREA Press.

Jifeng He and Huibiao Zhu. 2000. Formalising VERILOG. In *Proceedings of the 2000 7th IEEE International Conference on Electronics, Circuits and Systems, ICECS 2000, Jounieh, Lebanon, December 17-20, 2000*. IEEE, 412–415. https://doi.org/10.1109/ICECS.2000.911568

Yann Herklotz, James D. Pollard, Nadesh Ramanathan, and John Wickerson. 2021. Formal verification of high-level synthesis. *Proc. ACM Program. Lang.* 5, OOPSLA (2021), 1–30. https://doi.org/10.1145/3485494

Zhu Huibiao and He Jifeng. 2000. A DC-based Semantics for Verilog. In *Published in the proceedings of the ICS2000, Yulin Feng, David Notkin and Marie-Claude Gaudel (eds), Beijing*. Citeseer, 421–432.

Adam M. 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 2017, Irvine, CA, USA, November 13-16, 2017*, Sri Parameswaran (Ed.). IEEE, 209–216. https://doi.org/10.1109/ICCAD.2017.8203780

Matt Kaufmann, Panagiotis Manolios, and J Strother Moore. 2013. *Computer-aided reasoning: ACL2 case studies*. Vol. 4. Springer Science & Business Media.

Shriram Krishnamurthi. 2015. Desugaring in Practice: Opportunities and Challenges. In *Proceedings of the 2015 Workshop on Partial Evaluation and Program Manipulation* (Mumbai, India) *(PEPM '15)*. Association for Computing Machinery, New York, NY, USA, 1–2. https://doi.org/10.1145/2678015.2678016

Shriram Krishnamurthi, Benjamin S. Lerner, and Liam Elberty. 2019. The Next 700 Semantics: A Research Challenge. In *3rd Summit on Advances in Programming Languages, SNAPL 2019, May 16-17, 2019, Providence, RI, USA (LIPIcs, Vol. 136)*, Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 9:1–9:14. https://doi.org/10.4230/LIPIcs.SNAPL.2019.9

Andreas Lööw. 2022. A small, but important, concurrency problem in Verilog's semantics? (Work in progress). In *20th ACM-IEEE International Conference on Formal Methods and Models for System Design, MEMOCODE 2022, Shanghai, China, October 13-14, 2022*. IEEE, 1–6. https://doi.org/10.1109/MEMOCODE57689.2022.9954591

Andreas Lööw and Magnus O. Myreen. 2019. A proof-producing translator for verilog development in HOL. In *Proceedings of the 7th International Workshop on Formal Methods in Software Engineering, FormaliSE@ICSE 2019, Montreal, QC, Canada, May 27, 2019*, Stefania Gnesi, Nico Plat, Nancy A. Day, and Matteo Rossi (Eds.). IEEE / ACM, 99–108. https://doi.org/10.1109/FormaliSE.2019.00020

Xingyu Meng, Shamik Kundu, Arun K. Kanuparthi, and Kanad Basu. 2022. RTL-ConTest: Concolic Testing on RTL for Detecting Security Vulnerabilities. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* 41, 3 (2022), 466–477. https://doi.org/10.1109/TCAD.2021.3066560

Patrick O'Neil Meredith, Michael Katelman, José Meseguer, and Grigore Rosu. 2010. A formal executable semantics of Verilog. In *8th ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE 2010), Grenoble, France, 26-28 July 2010*. IEEE Computer Society, 179–188. https://doi.org/10.1109/MEMCOD.2010.5558634

José Meseguer and Grigore Rosu. 2007. The rewriting logic semantics project. *Theoretical Computer Science* 373, 3 (2007), 213–237. https://doi.org/10.1016/j.tcs.2006.12.018

Terence Parr. 2013. The definitive ANTLR 4 reference. *The Definitive ANTLR 4 Reference* (2013), 1–326.

Joe Gibbs Politz, Matthew J. Carroll, Benjamin S. Lerner, Justin Pombrio, and Shriram Krishnamurthi. 2012. A Tested Semantics for Getters, Setters, and Eval in JavaScript. In *Proceedings of the 8th Symposium on Dynamic Languages* (Tucson, Arizona, USA) *(DLS '12)*. Association for Computing Machinery, New York, NY, USA, 1–16. https://doi.org/10.1145/2384577.2384579

Joe Gibbs Politz, Alejandro Martinez, Mae Milano, Sumner Warren, Daniel Patterson, Junsong Li, Anand Chitipothu, and Shriram Krishnamurthi. 2013. Python: The Full Monty. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications* (Indianapolis, Indiana, USA) *(OOPSLA '13)*. Association for Computing Machinery, New York, NY, USA, 217–232. https://doi.org/10.1145/2509136.2509536

Fabian Schuiki, Andreas Kurth, Tobias Grosser, and Luca Benini. 2020. LLHD: A Multi-Level Intermediate Representation for Hardware Description Languages. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 258–271. https://doi.org/10.1145/3385412.3386024

Wilson Snyder. 2023a. *Verilator*. https://veripool.org/verilator/documentation/

Wilson Snyder. 2023b. *vppreproc*. https://metacpan.org/dist/Verilog-Perl/view/vppreproc

Lenny Truong and Pat Hanrahan. 2019. A Golden Age of Hardware Description Languages: Applying Programming Language Techniques to Improve Design Productivity. In *3rd Summit on Advances in Programming Languages, SNAPL 2019, May 16-17, 2019, Providence, RI, USA (LIPIcs, Vol. 136)*, Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 7:1–7:21. https://doi.org/10.4230/LIPIcs.SNAPL.2019.7

Viktor Vafeiadis, Thibaut Balabonski, Soham Chakraborty, Robin Morisset, and Francesco Zappa Nardelli. 2015. Common Compiler Optimisations Are Invalid in the C11 Memory Model and What We Can Do about It. In *Proceedings of the*

42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (Mumbai, India) *(POPL '15)*. Association for Computing Machinery, New York, NY, USA, 209–220. https://doi.org/10.1145/2676726.2676995

Stephen Williams. 2023. *Icarus Verilog*. https://steveicarus.github.io/iverilog/

Hasini Witharana, Yangdi Lyu, Subodha Charles, and Prabhat Mishra. 2022. A Survey on Assertion-Based Hardware Verification. *ACM Comput. Surv.* 54, 11s, Article 225 (sep 2022), 33 pages. https://doi.org/10.1145/3510578

Huibiao Zhu, Jifeng He, and Jonathan P. Bowen. 2008. From algebraic semantics to denotational semantics for Verilog. *Innov. Syst. Softw. Eng.* 4, 4 (2008), 341–360. https://doi.org/10.1007/s11334-008-0069-9

Received 2023-04-14; accepted 2023-08-27