A small, but important, concurrency problem in Verilog's semantics? (Work in progress)

Andreas Lööw
Imperial College London
London, UK

***Abstract*—Despite its many flaws, Verilog is today both the most popular hardware design language and a popular language for communication between hardware development tools. Ever since the language was standardised, researchers have made attempts at formalising its semantics. To this day, no such attempt has been fully successful. In this paper, we highlight one – we think, important – concurrency problem in Verilog's semantics that has, for now, sidetracked our own ongoing Verilog semantics formalisation attempt. To us, the problem calls for a clarification of the Verilog standard. We propose a potential fix for the problem.**

***Index Terms*—Verilog, semantics, concurrency**

I. Introduction

In the ideal case, formalising the semantics of a software or hardware language is a simple matter of first (1) carefully reading the language's standard and then (2) writing down what was read, in some appropriate mathematical form. Sadly, life is rarely this easy. E.g., in their C formalisation project, Memarian et al. [1], [2] end up balancing multiple *sources of truth*, including the C standard, existing C code, experimental data from compilers, and beliefs held by systems programmers and compiler writers (which Memarian et al. collect by a survey).

In this work-in-progress paper, we report on our work towards a formalisation of (a subset of) the hardware description language Verilog. The end goal of our work is a detailed formal semantics for the language, closely following the Verilog standard [3]. In this paper, specifically, we contribute a discussion on a problem, which we refer to as THE PROBLEM, at the core of Verilog's concurrency semantics. We offer an analysis of THE PROBLEM, and, in the process, we discuss some of the balancing acts between different sources of truth involved in formalising a hardware language. The two main sources of truth we consider are *the Verilog standard* and *the semantics of hardware*, but at times other sources come into play as well. Although not yet fully convinced of the correct way forward, we suggest a fix for THE PROBLEM that looks reasonable to us.

II. The problem

We now introduce THE PROBLEM, by example. Consider the (slightly contrived) module `netassign` in Fig. 1, a variant of Meredith et al.'s [4] `netassign` module (which we return to later). We ask: what should the value of `w` be after initialisation?

In the `netassign` module, the **`always`** block represents combinational logic, i.e., stateless logic, and we can think of the **`initial`** block as representing sequential logic, i.e., stateful logic. In Verilog, the behaviour of blocks such as **`always`** and **`initial`** is defined by C-like code ("behavioural code") and statements inside a block are executed in order.

```
module netassign;
 logic w, r;

 // always block
 always @(r)
  w = r;

 // initial block
 initial begin
  r = 0;
  r = 1;
 end
endmodule
```

Fig. 1: Example module illustrating THE PROBLEM.

The semantics of hardware tells us the answer to the `netassign` question is that `w` must be equal to `1` after initialisation since the **`always`** block is supposed to model combinational logic (and hence have no memory). Thus, to properly model hardware, Verilog's semantics must give the same behaviour. However, as *the Verilog standard* is written now, it appears to not be the case.

III. A short introduction to Verilog's semantics

We now give a short introduction to Verilog's semantics. The introduction we give, in particular the semantics of processes, is intentionally vague because the exact details of how processes are to be executed are unclear. Indeed, the very purpose of this paper is to highlight an unclarity at the centre of the semantics of processes in Verilog.

The Verilog standard defines a scheduling semantics for Verilog [3, Ch. 4], commonly referred to as Verilog's *simulation semantics*. Verilog's *synthesis semantics*, as defined by the (now withdrawn) synthesis standard [5], is relevant here only insofar as it tells us what kind of logic different constructs are mapped to (combinational vs. sequential logic).

Verilog's simulation semantics is event based and divides execution into simulation cycles. In short, Verilog allows hardware designers to, in various ways, model hardware as concurrently executing shared-memory processes. Processes can be defined by e.g., as we have just seen in the `netassign` example, **`initial`** blocks and **`always`** blocks; the former block type executes just once and the latter executes repeatedly.

978-1-5386-6195-6/18/$31.00 ©2022 IEEE

The simulation semantics is event based in the sense that the different kinds of Verilog constructs available induce processes that react/wait for different relevant events, e.g., an `always` block modelling sequential logic might be configured to react to positive edges of a circuit's clock signal. E.g., the following `always` block increments two registers `a` and `b` at every positive edge of a clock `clk`:[1]

```
always @(posedge clk) begin
 a <= a + 1;
 b <= b + 2;
end
```

Synthesisable `always` blocks have only one event control construct, here `@(posedge clk)`, which must be placed at the very beginning of the block. Nonsynthesisable `always` blocks, such as test bench code, are under no such restriction.

Blocks can also represent combinational logic:

```
always @(a, b)
 c = a + b;
```

The above block reacts to updates on the inputs `a` and `b` instead of clock edges. For combinational logic, the event control's so-called sensitivity list – in the above example, `a, b` – must include all inputs the block depends on if the block is to be synthesisable; nonsynthesisable blocks have no such restriction.

At the core of Verilog's simulation semantics, we find its stratified event queue and event scheduling algorithm. We do not, however, detail them in this paper. For our purposes, it is enough to know that some constructs, such as assignments, when executed, generate events other processes are able to react to, by event control constructs, as illustrated above.

IV. The solution?

We now return to the `netassign` module (Fig. 1). We first offer our understanding of the problem illustrated by the module and then suggest a fix.

A. Analysis of the problem

To us, the core of THE PROBLEM is that processes cannot be (1) expected to react too *all* (relevant) events – as required by the semantics of hardware – and at the same time be treated as (2) nondeterministically interleaved software-like threads – as required by the Verilog standard. In other words, our two main sources of truth are in conflict!

Regarding (2), the standard [3, p. 66] is clear on the point that an executing process can be preempted at any time:[2]

> Another source of nondeterminism is that statements without time control constructs in procedural blocks do not have to be executed as one event. [...] At any time while evaluating a procedural statement, the simulator may suspend execution and place the partially completed event as a pending event in the event region. The effect of this is to allow the interleaving of process execution, although the order of interleaved execution is nondeterministic and not under control of the user.

Concretely, for the `netassign` module, THE PROBLEM plays out as follows. For the purpose of illustration, say processes can be in one of two separate states: they are either running or they are waiting for one or more events to happen. If a process is in its running state, it is executed along with the other running processes in an interleaved manner until it reaches its event control (i.e., the start of its block). Now, the following interleaving poses a problem: (1) Both processes start in their running state. (2) The `always` block starts first and immediately enters its waiting state because of `@(r)`. (3) The `r = 0` write from the `initial` process executes, and this notifies the `always` process that `r` has been updated; consequently, the `always` process is now running again. (4) `w = r` from the `always` process execute. (5) `r = 1` from the `initial` process executes, which the `always` process does not see since it is in its running state (i.e., it is not listening for events). (6) The `initial` process finishes and the `always` process enters its waiting state again.

Ultimately, of course, THE PROBLEM is not limited to combinational-logic processes (like in the `netassign` module), stateful process are equally affected.

B. Suggestion to address the problem

We suggest a minimally invasive (and therefore, hopefully, largely backward compatible) fix to THE PROBLEM: restrict interleavings by disallowing preemption (i.e., modify (2)). That is, after being woken up by an event, a process executes without interruption until it hits an event control (or some other blocking construct). The same approach is taken by Gordon [7] in his work on a Verilog-inspired language, and as we shall soon see, in some of the Verilog tools in use today.

V. How open-source tools handle the problem

We now investigate how open-source Verilog tools handle THE PROBLEM. In other words, we address the question of how the tools cope with implementing a seemingly broken language standard. Specifically, of relevance is simulation tools, not synthesis tools, since we are investigating Verilog's simulation semantics, not Verilog's synthesis semantics. Therefore, we do not consider open-source synthesis tools such as Yosys [8].

In total, we consider two simulation tools: Icarus Verilog [9] and Verilator [10].

A. Icarus

Icarus is an event-based simulator that, in our impression, follows the standard closely. Icarus handles THE PROBLEM

[1] Note that so-called nonblocking assignments (`<=`) are used instead of so-called blocking assignments (`=`) in the example. We do not discuss the differences between the two types of assignments in this paper since the differences do not affect our discussion.

[2] The standard, however, never, as far as we can tell, clarifies the exact nature of these interleavings/preemptions. (E.g., are statements atomic?) With that said, effectively, we think, for a comparison with software interleavings, the memory consistency model Verilog ends up with is something similar to sequential consistency [6]. E.g., processes always read the latest value written to a variable [3, p. 85] (i.e., straightforward memory visibility guarantees) and statements inside blocks execute in order [3, p. 209] (i.e., no statements are reordered). (For synchronous synthesisable code, many memory-model questions are rendered trivial since (with a few exceptions, like multi-driven nets and block-memory modelling) there is at most one process writing to a variable/net (more precisely, no writes to overlapping "longest static prefixes" [3, p. 282] of variables/nets) and processes can communicate in a race-free manner using nonblocking assignments and the shared clock.)

```
module loop;
 logic v = 0;

 // Process 1
 initial forever v = 0;

 // Process 2
 initial $display("now i'm here");
endmodule
```

Fig. 2: Verilog module `loop`. (If using `always ...` instead of `initial forever ...`, Icarus aborts with an error complaining that the block will loop infinitely.)

by not doing thread preemption; a comment above the top-level process simulation function `vthread_run` [11] states: "Cause this thread to execute instructions until in [sic] is put to sleep by executing some sort of delay or wait instruction." (As an aside, discussions about the kind of interleavings we discuss in this paper appear to have taken place on the Icarus mailing list [12], and those discussions in turn refer to previous relevant discussions on similar topics.)

The way thread scheduling is implemented in Icarus is sensitive to the declaration order of the processes. E.g., consider the (nonsynthesisable) module `loop` in Fig. 2. If simulated as given in Fig. 2, the `loop` module never prints "now i'm here" (since process 1 never is preempted). If the order of the two processes is changed in the Verilog source code, "now i'm here" is printed immediately. With preemption, we would not have the same behaviour as in Icarus. It is, however, unclear what behaviour to prefer here (as we do not have the semantics of hardware as a guiding light since the module is nonsynthesisable).

In conclusion, Icarus leaves us with the impression that not doing preemption is a viable option for real-world tools since Icarus is a widely used tool and the `vthread_run` comment is (according to `git blame`) over 20 years old. (Of course, a simulator is free to not preempt threads even if allowed by the language standard since a simulator is not forced – nor expected – to explore all possible execution paths. Nevertheless, the existence of a nonpreempting simulator tells us that not doing preemption is viable in practice.)

B. Verilator

We now turn to the simulator Verilator. The simulation approach taken in Verilator – synthesis to C++ (or SystemC) – has more in common with synthesis tools than event-driven simulators like Icarus. Moreover, our impression is that the Verilator development team is more than happy to trade standard conformity for speed (e.g., Verilator is based on two-state simulation rather than the standard's four-state). Hence, Verilator does not help us in resolving THE PROBLEM.

VI. Previous work

We now relate our work to previous work. For context, the first Verilog standard was Verilog-1995 [13] and the most recent standard is SystemVerilog-2017 [3]. (Our discussions in this paper are based on the latest standard.)

The following is an, in all likeliness nonexhaustive, enumeration of previous Verilog formalisation projects (but nevertheless includes all previous projects we are aware of):

- Gordon[3] [14] (1995)
- Schneider and Xu [15], [16] (1998)
- Pace's PhD thesis work [17] (1998)
- Jifeng et al.'s long line of work, e.g., [18]–[23] (starting 2000)
- Pace [24] (2000) plus references (see also Gordon [7])
- Meredith et al. [4] (2010)
- Lööw and Myreen [25] (2019)

Meredith et al.'s [4] Verilog semantics, implemented in the K framework, is, to the best of our knowledge, the most complete and detailed Verilog semantics to date. (The semantics is based on Verilog-2005 [26].) When comparing our work to previous work, we therefore focus foremost on their semantics. Other previous projects, in our view, provide not a direct formalisation of the standard, but rather provide alternative Verilog semantics designed to be usable for reasoning or otherwise aid our understanding of Verilog as a language. These aims in themselves are well worth pursuing; but ultimately, any nonstandard semantics must be related back to (a formalisation of) the standard semantics of Verilog to show the nonstandard semantics, in some sense, sound with respect to the standard semantics. In other words, whatever one's aims with one's Verilog semantics is, a formalisation of the standard semantics will, eventually, be needed.

A. Meredith et al.'s Verilog semantics

The `netassign` module in Fig. 1 is a variant of Meredith et al.'s `netassign` module, we now discuss Meredith et al.'s analysis of their module.

Meredith et al.'s `netassign` module uses a continuous assignment instead of an `always` block for combinational modelling (i.e., `assign w = r`) and, in their module, `w` is declared a net (specifically, a `wire` net) instead of a variable. We now discuss the two approaches to the semantics of continuous assignments they discuss and highlight some problems with their discussions. Lastly, we relate the discussion back to THE PROBLEM.

Meredith et al.'s first approach is to give continuous assignments a procedural-assignment-like semantics (i.e., a semantics similar to the semantics of assignments made inside e.g. `always` blocks) since "[t]he best [Meredith et al.] can glean from the standard is that a [continuous] assignment should perform essentially as an `always` block with one blocking assignment in it [...]."[4] This approach results in problems for them since under this approach `0` is one possible value for `w` after initialisation (which, as pointed out in Sec. II, is not consistent with the semantics of combinational logic). To

[3]Only includes an informal semantics/outline for a formal semantics

[4]For readers familiar with Verilog: Clearly, this "glean" of the standard holds only in a limited range of situations. E.g., if a net is driven by multiple continuous assignments, the resultant net value is determined by the resolution function of the net type, which is not the case for variables written to by multiple `always` blocks [3, p. 85].

Meredith et al., the problem is that there should be at most "one outstanding update event [in the event queue] for a given net at a time." To address this problem, they suggest to enforce a programming style where "[multiple] assignments to the same variable should never occur in the same simulator cycle". We see this as too invasive, given e.g. how common the practice is of assigning default values to variables in the beginning of procedural blocks. (See Sec. A for an example.) In other words, the suggested approach goes against the source of truth provided by existing Verilog code.

Meredith et al.'s second approach, to avoid multiple outstanding update events, brings in the Verilog standard's semantics of continuous assignments into the discussion. However, instead of citing the standard, they cite Gordon [14]. They say that their discussion takes inspiration from Gordon's discussion on "cancelling". The idea is that new update events from continuous assignments cancel previous update events in the event queue, thereby avoiding ending up with multiple outstanding update events. Meredith et al. see their discussion as addressing a flaw in the standard; however, cancelling is part of Verilog standard's behaviour of continuous assignments.[5] Specifically, if we understand continuous assignments without specified delays as continuous assignments with zero delay, then "descheduling" as described in Sec 10.3.3 "Continuous assignment delays" in the standard (Sec. 6.1.3 in the Verilog-2005 standard their formalisation is based on), in our reading, applies here.

Given the semantics of continuous assignments, it is reasonable to ask: could we, instead of dropping preemption, address THE PROBLEM by adopting continuous-assignment-like semantics for procedural assignments? We say "no". Consider e.g. a block `always @(inp1, inp2) w = inp1 + inp2;`. Given the problem interleaving in Sec. IV-A, it is clear that such a block can miss events even without there ever being multiple assignments to the same variable/net in a simulation cycle.

VII. What about VHDL?

Verilog is often compared to (its archenemy) VHDL, defined by the VHDL-2019 standard [27]. Given the many similarities between the two languages, it is reasonable to ask: does THE PROBLEM occur in VHDL? We believe the answer to be "no".

According to Sec. 14.7.5 "Model execution" of the VHDL standard, after a process has been "resumed" as a result of an event occurring on an input the process is currently sensitive to, "[t]he process executes until it suspends." This might leave the reader with the impression that interleavings between processes are not part of VHDL execution. However, Sec. 11 "Concurrent statements" (a process statement is a type of concurrent statement) of the VHDL standard states the following:

> Within a given simulation cycle, an implementation may execute concurrent statements in parallel or in some order. The language does not define the order, if any, in which such statements will be executed. A description that depends upon a particular order of execution of concurrent statements is erroneous.

The above might instead leave the reader with the impression that processes can be interleaved ("execute [...] in parallel"); but, at the same time, since a hardware design ("a description") whose behaviour (or correctness, depending on how you read the standard) depends on how its statements are scheduled is "erroneous", interleavings cannot have any effect on executions.

A second source of truth we can consider is previous work on formalising the semantics of VHDL. E.g., Van Tassel's [28] formal VHDL semantics, for VHDL-1987 [29], which is "defined in an operational manner that closely adheres to the informal description of the simulation model of full VHDL" (p. 104), does not include interleavings (see "cs_3" on p. 93).

Clearly, regardless of whether VHDL allows for interleavings, the potentially problematic case is when interleavings are allowed. But even if we come to the conclusion that interleavings are part of VHDL executions, because of, in contrast to Verilog, processes reacting to events and processes executing are split into two separate phases in VHDL executions (see, again, Sec. 14.7.5 "Model execution"), interleavings do not, as far as we can tell, pose a problem for VHDL in the way we have, in this paper, shown they pose a problem for Verilog. As, when reacting to events are, as in VHDL, handled by a phase separate from running processes, no process will ever miss events as a result of being in its running state, since running happens in a separate running phase.

Hence, we conclude, even if VHDL and Verilog share many similarities, as the cores of their simulation algorithms are fundamentally different from each other, considering THE PROBLEM in the context of VHDL does not help us in addressing THE PROBLEM in the context of Verilog.

VIII. Conclusion

We have, based on our reading of the Verilog standard, highlighted a concurrency-related problem in the semantics of Verilog. At the very least, if our reading of the standard is somehow flawed, we have shown an opportunity to clarify parts of the standard. If, on the other hand, our reading is correct, substantial questions remain:

- Here, we have focused on synthesisable fragments of Verilog. Does our suggestion to drop interleavings hold up if we also considered nonsynthesisable fragments? E.g., (the most straightforward form of) busy waiting is impossible without preemption – but, at the same time, this should not be a problem considering Verilog's rich support for waiting for events.
- How does ruling out nondeterministic interleavings between processes affect important simulator (and synthesis tool) optimisations? Are, suddenly, important optimisations no longer allowed? (See Sec. B for a short comment.)

Finally, when THE PROBLEM has been settled, we can continue our work towards an event-driven formal semantics for Verilog based on a detailed understanding of Verilog's simulation semantics, which can be used, among other things, to validate previous simplified Verilog semantics projects.

[5] Indeed, Gordon [14] himself seems to introduce cancelling to be faithful to (his reading of) the standard: "Verilog's semantics specifies that at most one change to a given wire can be scheduled at any one time, [...]."

APPENDIX

A. Multiple assignments example

The following code is example 4.5c "case with defaults listed before case statement" from Mills [30], included here for convenience:

```
always_comb begin
 out1 = in1a;
 out2 = in2a;
 case (sel)
  cond2: out2 = in2b;
  cond3: out1 = in1c;
 endcase
end
```

Note how it is possible for both `out1` and `out2` to be assigned multiple times in the same simulation cycle. (An **`always_comb`** block is a variant of an **`always`** block with an automatically inferred sensitivity list.)

B. A short comment on interleaving-based optimisations

```
module redundant;
 logic a, b, c, inp;

 always_comb begin
  $display("EVAL 1: time = %0d, inp = %b, b = %b",
           $time, inp, b);
  a = inp;
  c = b + 1;
 end

 always_comb begin
  $display("EVAL 2: time = %0d, a = %b",
           $time, a);
  b = a;
 end

 initial begin
  // "delayed" assignments, causes the process to
  // suspend for 5 simulation cycles before continuing
  #5 inp = 1;
  #5 inp = 0;
 end

endmodule
```

Fig. 3: Verilog module `redundant`.

Here, we cannot do justice to the question of how allowing arbitrary interleavings and simulator optimisations are connected. We can, however, make some preliminary remarks.

Consider the (nonsynthesisable) module in Fig. 3. If arbitrary interleavings are allowed, a smart simulator could schedule the processes such that they execute in the same order as the following block:

```
always_comb begin
 a = inp;
 b = a;
 c = b + 1;
end
```

To investigate whether today's simulators take advantage of such opportunities, as allowed by the current Verilog standard, we ran the module in Fig. 3 using Icarus 11.0 and the four commercial simulators available on EDA playground (https://edaplayground.com), which are, as of the time of this writing, Aldec Riviera Pro 2020.04, Cadence Xcelium 20.09, Mentor Questa 2021.3, and Synopsys VCS 2020.03. All the simulators give the same printout (modulo the order of execution in time 0):

```
EVAL 1: time = 0, inp = x, b = x
EVAL 2: time = 0, a = x
EVAL 1: time = 5, inp = 1, b = x
EVAL 2: time = 5, a = 1
EVAL 1: time = 5, inp = 1, b = 1
EVAL 1: time = 10, inp = 0, b = 1
EVAL 2: time = 10, a = 0
EVAL 1: time = 10, inp = 0, b = 0
```

Note how all simulators execute the first **`always_comb`** block twice both at time 5 and 10 (instead of interleaving the two blocks such that no redundant re-execution is needed).

The behaviour of Icarus is expected given that Icarus does not do process preemption. What the four commercial simulators are doing behind the scenes we cannot know since the simulators are not open source. We can, however, draw the conclusion that none of the simulators – for this module – take advantage of the additional optimisation opportunities enabled by arbitrary interleavings. This, of course, does not give us an answer for the general case, but nevertheless indicates that dropping process preemption might be a viable way forward.

REFERENCES

[1] K. Memarian, J. Matthiesen, J. Lingard, K. Nienhuis, D. Chisnall, R. N. M. Watson, and P. Sewell, "Into the depths of C: Elaborating the de facto standards," in *Conference on Programming Language Design and Implementation (PLDI)*, 2016.

[2] K. Memarian, V. B. F. Gomes, B. Davis, S. Kell, A. Richardson, R. N. M. Watson, and P. Sewell, "Exploring C semantics and pointer provenance," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, 2019.

[3] "IEEE standard for SystemVerilog–unified hardware design, specification, and verification language," *IEEE Std 1800-2017*, 2018.

[4] P. Meredith, M. Katelman, J. Meseguer, and G. Roşu, "A formal executable semantics of Verilog," in *International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, 2010.

[5] "Verilog register transfer level synthesis," *IEEE Std 62142-2005*, 2005.

[6] L. Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs," *IEEE Transactions on Computers*, vol. C-28, no. 9, 1979.

[7] M. J. C. Gordon, "Relating event and trace semantics of hardware description languages," *The Computer Journal*, vol. 45, no. 1, 2002.

[8] C. Wolf, "Yosys open synthesis suite." [Online]. Available: https://yosyshq.net/yosys

[9] "Icarus verilog web page." [Online]. Available: http://iverilog.icarus.com

[10] "Verilator web page." [Online]. Available: https://veripool.org/verilator

[11] "Source file `vthread.h` from Icarus 'stable version 11.0' source code," 2020. [Online]. Available: https://github.com/steveicarus/iverilog/blob/v11_0/vvp/vthread.h#L66

[12] "Mailing list iverilog-devel: Verilog execution of always block as multiple events," 2020. [Online]. Available: https://sourceforge.net/p/iverilog/mailman/iverilog-devel/thread/2043773330.394897.1589755112783@mail.yahoo.com

[13] "IEEE standard hardware description language based on the Verilog hardware description language," *IEEE Std 1364-1995*, 1996.

[14] M. Gordon, "The semantic challenge of Verilog HDL," in *Symposium on Logic in Computer Science*, 1995.

[15] G. Schneider and Q. Xu, "Towards a formal semantics of Verilog using duration calculus," in *Formal Techniques in Real-Time and Fault-Tolerant Systems*, 1998.

[16] G. Schneider and X. Qiwen, "Towards an operational semantics of Verilog," International Institute for Software Technology, United Nations University, Tech. Rep. 147, 1998.

[17] G. J. Pace, "Hardware design based on Verilog HDL," Ph.D. dissertation, Oxford University, 1998.

[18] H. Jifeng and X. Qiwen, "An operational semantics of a simulator algorithm," International Institute for Software Technology, United Nations University, Tech. Rep. 204, 2000.
[19] H. Jifeng and Z. Huibiao, "Formalising Verilog," in *International Conference on Electronics, Circuits and Systems (ICECS)*, 2000.
[20] J. P. Bowen, H. Jifeng, and X. Qiwen, "An animatable operational semantics of the Verilog hardware description language," in *International Conference on Formal Engineering Methods (ICFEM)*, 2000.
[21] H. Zhu, J. He, and J. P. Bowen, "From operational semantics to denotational semantics for Verilog," in *Correct Hardware Design and Verification Methods (CHARME)*, 2001.
[22] Z. Huibiao, J. P. Bowen, and H. Jifeng, "Deriving operational semantics from denotational semantics for Verilog," in *Proceedings Eighth Asia-Pacific Software Engineering Conference*, 2001.
[23] H. Zhu, J. He, and J. P. Bowen, "From algebraic semantics to denotational semantics for Verilog," in *International Conference on Engineering of Complex Computer Systems (ICECCS)*, 2006.
[24] G. J. Pace, "The semantics of Verilog using transition system combinators," in *Formal Methods in Computer-Aided Design (FMCAD)*, 2000.
[25] A. Lööw and M. O. Myreen, "A proof-producing translator for Verilog development in HOL," in *Formal Methods in Software Engineering (FormaliSE)*, 2019.
[26] "IEEE standard for Verilog hardware description language," *IEEE Std 1364-2005*, 2006.
[27] "IEEE standard for VHDL language reference manual," *IEEE Std 1076-2019*, 2019.
[28] J. P. Van Tassel, "An operational semantics for a subset of VHDL," in *Formal Semantics for VHDL*, C. D. Kloos and P. T. Breuer, Eds. Springer, 1995.
[29] "IEEE standard VHDL language reference manual," *IEEE Std 1076-1987*, 1988.
[30] D. Mills, "Yet another latch and gotchas paper," in *Synopsys Users Group Conference (SNUG)*, 2012.