Visually Debugging the Semantics of Verilog

Andreas Lööw

Imperial College London, UK

Abstract. Verilog's simulation semantics, defined by the Verilog standard, has thus far eluded definitive mathematical formalisation. In this paper, to prepare the ground for such a formalisation, we employ an, what we believe to be, underappreciated form of semantics formalisation: namely, visual formalisation. We develop a visual formalisation of Verilog's simulation semantics and use this formalisation to visually debug, i.e., visually find problems in, the standard's description of the semantics. We identify two problems in the Verilog standard that the current state-of-the-art mathematical formalisation projects for Verilog have either missed or, we argue, misunderstood. We believe the two identified problems (1) constitute actionable findings for improving the Verilog standard and (2) demonstrate the complementary value of visual formulation.

1 Introduction

To reason formally about a programming language or a hardware-description language (HDL), a mathematical formalisation (such as an operational semantics or denotational semantics) of the language is required. Examples of applications of formal reasoning include the verification of programs/hardware designs implemented in the language in question and the verification of tools for the language, such as compilers/synthesis tools and analysis tools (e.g., model checkers).

Here we are interested in the most popular HDL [27], Verilog, which unfortunately lacks a definitive mathematical formalisation. The semantics of Verilog, specifically its simulation semantics, is defined by the (System)Verilog standard, IEEE 1800-2017 [16]. As is common for language standards, the Verilog standard is written in prose form. The Verilog standard is infamous for being difficult to read and understand, which has proven formalisation of Verilog to be difficult [21, 22, 28, 32–34, 42, 47, 51–53, 62, 63]. As an HDL, the semantics of Verilog is distinctly different from the semantics of traditional programming languages such as C++ or Haskell. In short, the semantics is concurrent and event driven: it is driven by concurrent processes that create and react to events, such as a clock tick or a change in a circuit input. The semantics is centred around the maintenance of an event queue, which is used to keep track of and coordinate different events.

Having in mind the advantages of going through a round of testing/bug finding before attempting to formally verify a piece of software or hardware, we believe "debugging" language standards before attempting mathematical formalisation is well-invested energy. This is particularly true for language standards like the Verilog standard, which is known to be difficult to work with.

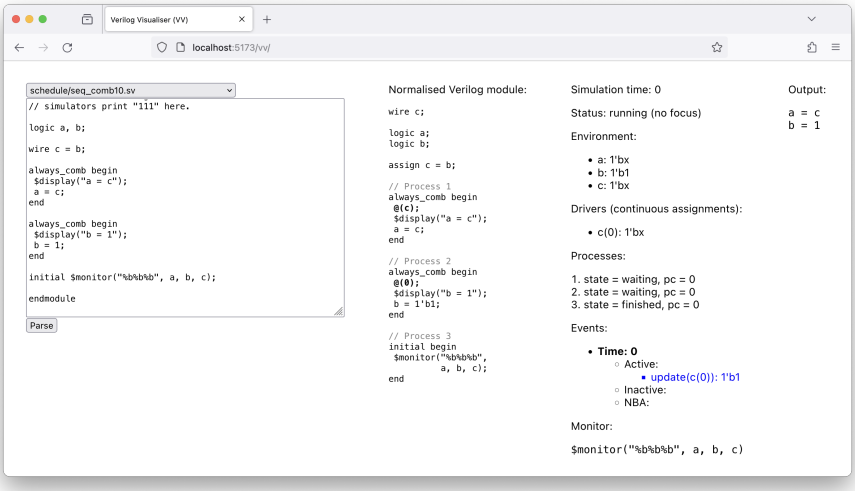

Fig. 1: A screenshot of VV. The third column (from left) visualises the current state of the Verilog event queue, as we explain in more detail, in Sec. 7, after having introduced Verilog's simulation semantics in more detail, in Sec. 4 and onwards.

In this paper, we employ *visual formalisation* to prepare the ground for a future definitive mathematical formalisation of Verilog: we visually debug the Verilog standard by visually formalising it and in the process we identify problems in the standard that provide actionable pathways to improve it. Previous work on mathematical formalisation of the standard have not found or have inadequately addressed the problems we have found, which we believe shows that *visual formalisation is a (perhaps, underappreciated) useful complement to traditional mathematical formalisation*. Specifically, we have developed a *visual formalisation of Verilog's simulation semantics* which has led us to identify *two problems in the Verilog standard* concerning the semantics of process interleavings and a core construct of Verilog called nonblocking assignments. We show how the identified problems manifest in the two state-of-the-art mathematical formalisations of Verilog, namely, Meredith et al. [42] and Chen et al. [22].

The visual formalisation of Verilog we have developed is a new *visual simulation tool*, called VV ("Verilog visualiser"). As we explain after introducing the semantics of Verilog in more detail, VV, shown in Fig. 1, visualises the structure and maintenance of Verilog's event queue and how the constructs of Verilog are given semantics in terms of their interactions with this queue. VV is interactive and driven by the user clicking the next event to execute, which has allowed us to visually explore different event schedules and other aspects of the semantics of Verilog. VV is web based, and the source code and a live demo of VV are available at `https://github.com/AndreasLoow/vv`.

We see our visual exploration of the standard which has identified two problems of the standard as the main contribution of this paper and see VV mainly as a by-product of this contribution. The structure of the paper reflects this attitude: the major chunk of the paper concerns the semantics of Verilog and the two problems. Specifically: in Sec. 2, we discuss what value visual formalisation can bring to the mathematical formalisation process. In Sec. 3, we introduce the subset of Verilog we consider. In Sec. 4, we describe what is needed of Verilog's simulation semantics to discuss the two problems we have found in the standard. In Sec. 5, we discuss the two problems we have found, and in Sec. 6, we highlight some nonproblems some readers will have expected to be problems for formalisation. In Sec. 7, we describe VV. We finish with related work in Sec. 8.

2 Formalising language standards

We lay out our view on developing and evaluating mathematical formalisations of language prose standards and where visual formalisation fits into this view.

Mathematical formalisations. For languages defined by a prose standard, the standard constitutes the ultimate authority on the semantics of the language. In previous work, formal methods have been used both to critique and/or improve existing standards, e.g., Bodin et al.'s [20] work on JavaScript, Memarian et al.'s [40, 41] work on C, and Klimis et al.'s [35] work on GPU computing, and to (less commonly) co-develop standards along with mathematical formalisations, e.g., the work on WebAssembly [30, 57, 58]. Work on Verilog formalisations fit into the former group, and the state-of-the-art formalisations are Meredith et al. [42], implemented in the K framework [50], and Chen et al. [22], presented by inference rules and also implemented in Java.

When working with an existing prose standard, a formalisation of the standard is successful if it in some sense "captures" and "corresponds" to the standard. Put pompously, there should be a simple and immediate ontological correspondence between the two (e.g., they should build on the same fundamental concepts). Less pompously, Bodin et al. [20], in their work on formalising JavaScript, suggests the term "eyeball closeness" for this formalisation-success criterion.

Developing mathematical formalisations. In theory, developing a mathematical formalisation of a prose standard is a straightforward activity: first, (1) carefully read the prose standard, and then, (2) carefully write down what was read in the form of a mathematical formalisation. In practice, however, the formalisation process is never as straightforward because of the inherent ambiguity and imprecision of prose text: we will inevitably run into omissions, gaps, and other problems of the standard. To overcome these problems, we cannot solely rely on the standard itself: instead, we must find support in a complex of sources that either explicitly or implicit provide their interpretation of the standard, while at the same time keeping in mind that the standard itself is the ultimate authority. This complex of sources helps us tease out a best-effort interpretation of the

standard such that we can write it down in mathematical form and/or critique the standard to improve it and prepare it for future formalisation work.

E.g., for their complex of sources, Bodin et al. [20] cite in their work on formalising JavaScript: the JavaScript prose standard, browser implementations of JavaScript, discussion groups such as es-discuss, and the official ECMA test suite test262. Memarian et al. [40, 41], in their work on formalising C, cite an even more eclectic complex of sources: the C prose standard, existing C code, experimental data from compilers, and surveys and interviews answers about C from systems programmers and compiler writers. Meredith et al. and Chen et al. in their Verilog work do not explicitly enumerate the sources they rely on beyond the Verilog standard. Meredith et al. seem to rely on existing Verilog simulators and (in one place in their discussion) the semantics of hardware. Chen et al. seem to rely on existing simulators and real-world Verilog code.

Evaluating mathematical formalisations. Evaluating whether a formalisation has correctly "captured" a prose standard is an inherently nonmathematical problem because a prose standard is a nonmathematical object. For evaluation, therefore, the best we can do is to poke and probe the formalisation, in more or less systematic ways.[1] E.g., we can compare the formalisation to other interpretations that can be found in the language's complex of sources: one common instance of this is to run it on a test suite, in effect comparing it with the implicit interpretation of the standard captured by the expected outcomes of the test cases of the test suite. Both Meredith et al. and Chen et al. evaluate by test suites. Some languages, like JavaScript, have an officially blessed test suite. This is not the case for Verilog. Instead, to our best understanding, Meredith et al. evaluate by running their semantics on a small collection of tests they have written themselves. Chen et al. evaluate by running the test suite of the Verilog simulator Icarus [2] and other real-world test suites (including a small RISC-V processor).

The role of visual formalisations. We believe that visual formalisations are useful members of languages' complex of sources and that they can help evaluate both mathematical formalisations and language standards themselves, for this paper, we focus on the latter. In our work for this paper, we have found that interactively running different handcrafted Verilog modules in our visual simulator has raised questions that have made us re-read and re-evaluate parts of the standard we previously thought we understood well. We hypothesise that, compared to mathematical formalisation, visual formalisation gives a complementary view of our understanding/interpretation of a standard, and hence raises different questions than mathematical formalisation. This hypothesis is corroborated by the fact that we have found two problems in the Verilog standard previous projects mathematically formalising the standard have not.

[1] It is of course possible to evaluate *other* properties of the formalisation by mathematical means, such as its coherence, by, e.g., proving meta-theoretical results about the formalisation. Nevertheless, to transport any of these properties to the prose standard, one will need an argument for why the formalisation correctly captures the standard.

3 Scope: target subset of Verilog

In this section, we introduce and discuss the subset of Verilog that we are interested in and that is supported by VV, i.e., our target subset of Verilog.

Target subset selection criteria. Verilog is a large language; the Verilog standard weighs in at 1315 pages – just the grammar alone occupies 45 pages (see App. A of the standard). We have identified a subset of Verilog we think is the most important and interesting to formalise, based on the following two criteria:

- *We focus on synthesisable Verilog*, which constitutes a relatively small subset of Verilog. Verilog is essentially two languages in one: synthesisable Verilog, used to describe the structure and behaviour of hardware designs, deriving its name from the fact it is the kind of Verilog synthesis tools accept, and nonsynthesisable Verilog, used to implement "test benches" for hardware designs, to provide stimuli-and-probe infrastructure.[2] We focus on synthesisable Verilog because our ultimate interest to pave the way for future formal reasoning for Verilog: in such reasoning, test benches are replaced by other stimuli-and-probe infrastructure. E.g., in model checking the stimuli-and-probe infrastructure might be an LTL or CTL formula.[3] With that said, we do also target a small select subset of nonsynthesisable Verilog, such that we can easily provide stimuli to hardware designs within VV.
- As we explore further in the next section, the semantics of Verilog is oriented around an event queue, and we are especially interested in exploring a representative subset of *core concurrency constructs* of Verilog that have *interesting semantics in terms of how they interact with this event queue*. In particular, Verilog contains many constructs for supporting programming-in-the-large [25] (in contrast to programming-in-the-small), such as metaprogramming constructs and modules, which are used to structure large hardware developments. These types of constructs are important in real-world code, but not for exploring the core concurrency semantics of Verilog, because such constructs do not complicate the event queue further.

The target subset. Fig. 2 describes the grammar of our target subset of Verilog, which we now discuss in bottom-up order. (Here, we describe the semantics of the subset only briefly and informally; we introduce (relevant parts of) the event-driven simulation semantics of Verilog in the next section.)

Values. Verilog bits can take on four different values [16, p. 83]. We include all four, see v (in Fig. 2). The value `x` is used for a multitude of purposes, often

[2] For readers not familiar with Verilog, we provide a small example circuit and test infrastructure in App. A to give a flavour of the difference between the two subsets.

[3] Even outside the domain of formal reasoning, various approaches to hardware development replace Verilog test benches with various other infrastructure. E.g., cocotb [1] enables implementing test benches in Python instead of (nonsynthesisable) Verilog.

n	$\in$	$\mathbb{N}$	s	::=	s `;` s — sequential sequencing
str	$\in$	strings		|	`if (`e`)` s [`else` s] — if statement
id	$\in$	identifiers		|	id `=` [`#`n] e — blocking assignment
v	::=	`0` | `1` | `x` | `z`		|	id `<=` [`#`n] e — nonblocking assignment
op_1	::=	`!` | `~` | $\cdots$		|	`@(`ee`)` s — event control
op_2	::=	`&` | `&&` | `+` | $\cdots$		|	`@(*)` s — combinational event control
e	::=	v		|	`#`n s — delay control
	|	id		|	`wait(`e`)` s — wait statement
	|	op_1 e		|	t`(`$te*$`)` — system task
	|	e op_2 e	m	::=	n | `(`n`,` n`)` | `(`n`,` n`,` n`)`
	|	e `?` e `:` e	nt	::=	`wire` | `wor` | `wand`
eee	::=	`edge` | `posedge`	pt	::=	`initial` | `final` | `always` | `always_ff`
	|	`negedge`		|	`always_comb` | `always_latch`
ee	::=	[eee] e | ee `or` ee	mi	::=	`logic` id [`=` e] — variable declaration
t	::=	`$display` \| `$monitor`		|	nt [`#`m] id [`=` e] — net declaration
	|	`$finish`		|	pt s — procedure/block
te	::=	e | str | `$time`		|	`assign` id `=` [`#`m] e — continuous assignment
			m	::=	$mi*$

Fig. 2: Values v, expressions e, statements s, module items mi, and modules m. Square brackets ([...]) denote optional elements and times ($\ldots*$) repetition. Redundant syntax is omitted to avoid clutter, e.g., "ee, ee" instead of "ee `or` ee" in event expressions (ee) or `reg` instead of `logic` in variable declarations (mi).

representing something like "unknown value", "invalid value", "error", or similar.[4] The value `z` represents a high-impedance state and is used to model tristate logic, as discussed below. We do not include array values, which are important for real-world code but do not complicate the Verilog event queue compared to only supporting bit values. Not including arrays has allowed us to avoid a long series of minor nonconcurrency problems we are not directly interested in for the moment, such as the implicit resizing semantics of Verilog.

Data objects. A data object in Verilog is a "named entity that has a data value and a data type associated with it" [16, p. 83]. There are two main groups of data objects in Verilog: variables and nets. We include both groups, see mi.[5] Variables have the same semantics as variables in imperative software languages, i.e., the last write to a variable determines its value. Nets have no analogue in software languages; the value of a net is determined by its set of "drivers", in our target subset, a set of continuous assignments, introduced below. More precisely,

[4] It is difficult to succinctly characterise precisely what role X values play in Verilog. E.g., Flake et al. [27] count eight different situations in where X values can arise.

[5] Verilog allows for implicitly declared nets; anecdotally, many consider implicit nets to be a misfeature that introduces more problems than it solves. To allow us to focus on the core concurrency semantics of Verilog rather than surface syntax problems, we require all nets to be explicitly declared. This can be understood as `default_nettype` being is set to `none` by default instead of `wire` [16, p. 685–686].

the value of a net is decided by the resolution function of the net type applied to the values of the drivers of the net. We include three types of nets: `wire`, `wor`, and `wand`. E.g., for a `wire` net, all drivers with value `z` are ignored in resolution, and all other drivers must have the same value, otherwise the net resolves to `x`. A `wand` net will, in contrast, resolve to the conjunction of the values of its drivers, e.g., with two drivers with the values `1` and `0`, the value of the net will be `0`.

Processes. Verilog is a process-based concurrent language. We include two types of processes: procedural processes and continuous assignments. See, again, *mi*.

Procedural processes are similar to software processes: they come with a program counter, internal state, etc. Processes induced by `initial` blocks execute once and terminate, whereas processes induced by `always` blocks execute over and over again in an infinite loop. The blocks `always_comb`, `always_ff`, and `always_latch` are variants of `always`; e.g., `always_comb` is an `always` block which executes when a data object it depends on changes. Expressions e and statements s include the usual imperative-software-language suspects, such as sequential sequencing and if statements. Statements s also include potentially less familiar faces to software audiences, such as the various timing- and event-control constructs used to coordinate process execution. We discuss these potentially less familiar faces further in the next section, when we discuss the Verilog event queue in more detail. Another peculiarity of Verilog is that procedural assignments are divided into blocking and nonblocking assignments, which we also discuss in the next section. Lastly, note that procedural processes can only write to variables, not nets, since they cannot participate in resolution.

The second type of process arise from continuous assignments `assign`. Continuous assignments can either function as a driver to a net or to a variable. If a net has multiple drivers, the values from the different drivers are merged using the resolution function of the net, as discussed above. A variable, in contrast, cannot have more than one driver. Processes induced by continuous assignments are not like software-language processes: the process associated with a continuous assignment is run every time a data object the assignment depends on is updated, the process does not have a program counter or other internal state.

Modules. The basic building block of Verilog circuits are modules, see m. For our work here, we consider one module at a time.

4 The simulation semantics of Verilog

Here, we discuss how the simulation semantics of Verilog is defined by the Verilog standard. Our aim here is not to introduce the full semantics, but to introduce enough of how the standard describes the semantics to be able to we discuss the two problems we have found in the standard while visually formalising it.

The Verilog standard defines the simulation semantics of Verilog by giving pseudocode for a "reference algorithm for simulation" [16, Sec. 4.5] in combination with prose text sprinkled throughout the standard. The reference algorithm is an

interpreter for Verilog, i.e., an operational semantics. The standard describes the reference algorithm at a high level and leaves the details of the algorithm to the imagination of its readers. The entry point of the algorithm is the following pseudocode function (all pseudocode in this section is from Sec. 4.5 of the standard):

```
execute_simulation {
 T = 0;
 initialize the values of all nets and variables;
 schedule all initialization events into time zero slot;
 while (some time slot is nonempty) {
  move to the first nonempty time slot and set T;
  execute_time_slot (T);
 }
}
```

The algorithm is oriented around events and maintains an event queue. The event queue is divided into "time slots". The variable `T` keeps track of the current the current time slot, or "simulation time". Each time slot is split into "regions". The following regions are relevant for our target subset of Verilog: active, inactive, NBA ("nonblocking assignment"), and observed.[6] Time slots are executed by the pseudocode function `execute_time_slot`. After restricting the function to the regions relevant here, the function becomes as follows:

```
execute_time_slot {
 while (any region in [Active ... Observed] is nonempty) {
  execute_region (Active);
  R = first nonempty region in [Active ... Observed];
  if (R is nonempty)
   move events in R to the Active region;
 }
}
```

That is, until all regions are empty, the events of the first nonempty region are moved to the active region and executed. The pseudocode function `execute_region` for executing regions is as follows:

```
execute_region {
 while (region is nonempty) {
  E = any event from region;
  remove E from the region;

  if (E is an update event) {
   update the modified object;
   schedule evaluation event for any process sensitive
   to the object;
  } else { /* E is an evaluation event */
   evaluate the process associated with the event and
```

[6] The standard includes 17 regions in total. The regions not included here are used to give semantics to constructs that are out of scope here, e.g., the Verilog APIs.

```
   possibly schedule further events for execution;
  }
 }
}
```

That is, the events of a region are executed in nondeterministic order and execution is driven by processes creating events – update events – and reacting to events – evaluation events. The details of executing individual events are described only by prose text. To exemplify some cases, we consider executing a process. Most of process execution happens in the active region. E.g., executing a nondelayed blocking assignment, say `y = 1`, updates the variable/net `y` and notifies other processes waiting in the active region for updates, e.g., a process wanting at a statement `@(posedge y)` if `y` was, e.g., `0` before. Executing a (nonzero-)delayed blocking assignment, say `#5 y = 1` or `y = #3 1`, schedules an event in a future time slot's active region relative to the current time. Zero-delayed statements, however, such as `#0 y = 1`, are scheduled in the inactive region of the current time slot. Nonblocking assignments also schedule events outside the active region: such assignments schedule update events in the NBA region (of the current time slot, or of a future time slot if the assignment is delayed). Scheduling events in the NBA region enables nonblocking assignments to be used for communication between processes since they do not race with events scheduled in the active region, such as most other aspects of process execution. Lastly, in our target subset of Verilog, the observed region is only used by the `$monitor` system task, to print values at end of time slots.[7]

5 Two problems of the Verilog standard

During our visualisation work for the Verilog standard, we have identified two problems of the standard by writing small Verilog test modules to try out corner cases of the semantics of Verilog and then interactively running them in VV until we saw something suspicious or unexpected occur, which caused us to investigate further. We now describe the two problems we have identified and moreover show that the current two state-of-the-art mathematical formalisations of Verilog, i.e., Meredith et al. [42] and Chen et al. [22], do not address the problems adequately.[8]

Interleaving of processes. The standard allows for too many interleavings between processes, breaking Verilog as a modelling language for hardware, as we now show. The problem arises from the following statement in the standard: "statements without time-control constructs in procedural blocks do not have to be executed as one event" [16, p. 66]. That is, the standard specifies that processes should

[7] The reader less familiar with Verilog might want to take a second look at App. A, containing a small circuit example and test bench, to see how the various constructs discussed here are used in practice.

[8] Meredith et al. and Chen et al. formalise Verilog-2005 [8] – not, as us, SystemVerilog-2017 [16]. The differences between the two standards do not matter for this section.

```
module interleave;

logic a, b;

always_comb
 b = a;

initial begin
 a = 0;
 a = 1;
end

endmodule
```

```
module nbinterleave1;

logic a;

always @(*)
 $display("%b", a);

initial begin
 a <= 0;
 a <= 1;
end

endmodule
```

```
module nbinterleave2;

logic a, b;

always @(*)
 $display("%b, %b", a, b);

initial begin
 a <= 1;
 b <= 1;
end

endmodule
```

Fig. 3: Three example modules

follow the usual interleaving semantics of processes from software languages (in its idealised form, known as, sequential consistency [38]). We discuss the key points of the problem here and provide further details in App. B, as referenced below.

Here, we illustrate the problem by a minimal example. In App. B, we provide an example consisting of a synthesisable module and a test bench, to show that the problem not only occurs in artificial examples. Now, consider the module `interleave` in Fig. 3. According to the Verilog synthesis conventions, the first process models combinational logic (that is, stateless logic). Therefore, to respect the semantics of hardware, `b` must equal `a` in quiescent states. Thinking of the two assignments in the second process as a fragment of another combinational block, we see that the following interleaving is not compatible with the semantics of hardware. (1) The second process executes `a = 0` and then preempts. (2) The first process executes `b = a` and then preempts. (3) The first process executes `a = 1` and terminates. (4) The first process then continues execution by going into waiting directly. There are now no more events to execute, but $\texttt{a} \neq \texttt{b}$. Effectively, the first process missed the second update to the variable `a`.

It is not immediately clear how to fix the standard. One simple candidate solution is to only allow processes to interleave when they block, however, this candidate solution is not without problems since there appear to be some corner cases of some of the existing Verilog simulators where processes are interleaved without blocking. We make some further comments on the behaviour implemented in existing simulators in App. B.

Both Meredith et al.'s and Chen et al.'s Verilog semantics follow the standard to the letter and allow for arbitrary interleavings. In particular, both semantics allow the interleaving sequence for `interleave` discussed above – which we show in App. B. Interestingly, since Chen et al. test their semantics against the test suite of the Icarus simulator, they note that 99 tests fail under their semantics because of (according to their own diagnosis) too many interleavings. Moreover, (some of) the real-world code they test only work under their semantics in the sense the test cases *can* pass, by running the test cases multiple times until an appropriate

schedule happens to be selected, rather than passing every run. Chen et al. claim this is because the test cases have insufficient synchronisation; in effect, claiming that current Verilog practice (as embodied by the Icarus test cases and the real-world code they test), rather than the standard, is incorrect. We do not see not changing the standard as realistic, given that, as we have shown above, even such simple code as the `interleave` module requires additional synchronisation when arbitrary interleavings are allowed (in this case, just because of a combinational block depends on other combinational block in the module). For the interested reader, we discuss one example from Chen et al.'s tests in App. B.

Semantics of nonblocking assignments. The pseudocode and the prose text for the semantics of nonblocking assignments are not consistent with each other: the function `execute_time_slot` of the pseudocode suggests that executing NBA events is a simple matter of moving all NBA events from the NBA region to the active region, however, the prose text of the standard suggests that executing NBA events is more involved. We have found in total two inconsistencies. Claims made below about Meredith et al.'s and Chen et al.'s semantics are backed up by examples in App. C.

Semantics of nonblocking assignments: first inconsistency. A defining characteristic of the active region is that its events are allowed to execute in any order, hence, doing what the pseudocode suggest and simply move all NBA events to the active region does not guarantee any order between the NBA events when executed. However, this is inconsistent with the prose text, which provide order guarantees. Moreover, making matters worse, the prose text is inconsistent with itself. On page 66 of the standard, it is said that NBA events, unconditionally, "shall be performed in the order the [nonblocking assignments] were executed", and on page 239 it is said "[t]he order of the execution of distinct nonblocking assignments to a given variable shall be preserved", i.e., only the order of NBA events *to the same variable* is required to be preserved.

Neither Meredith et al. nor Chen et al. call out these inconsistencies. Both their semantics maintain the order of all NBA events. When moving NBA events to the active region, Meredith et al. maintain the order dependencies by grouping all NBA events in the NBA region into a group event containing all the NBA events and move this group event to the active region instead of each individual NBA event. Chen et al. take a bigger step away from the pseudocode and maintain the order of the NBA events by spawning a new procedural process containing all writes of the NBA region instead of moving events between regions; we have found no support for this behaviour in the standard.

Semantics of nonblocking assignments: second inconsistency. The prose text of the standard enforces another order constraint of NBA events not enforced by the pseudocode. The following prose text can be found on page 238 of the standard:[9]

[9] Another unrelated problem: note that the quoted text has not been updated for SystemVerilog, specifically, assignments in the reactive region set execute after as-

> [...] the nonblocking assignments are the last assignments executed in a time step—with one exception. Nonblocking assignment events can create blocking assignment events. These blocking assignment events shall be processed after the scheduled nonblocking events.

Clearly, the authors of the standard did not intend the execution of blocking and nonblocking assignments to be interleaved. As far as we can tell, the standard does not comment on other types of events mixing with NBA events. E.g., is the execution of the `$display` statement in `nbinterleave1` in Fig. 3 allowed to mix with the execution of the nonblocking assignments? Our best *guess* is that the authors intended the same restriction to be true for other types of events and types of statements as well. Interestingly, for the subset of Verilog we consider here, not allowing NBA events to mix with other events leaves no externally observable difference between maintaining the order of all NBA events vs. the order of NBA events for the same variable. To exemplify, consider `nbinterleave2` in Fig. 3 when NBA events are not allowed to mix with other events.

Neither Meredith et al. nor Chen et al. discuss the above order constraint. Neither enforce it in their semantics.

6 Nonproblems of the Verilog standard

Verilog, as it deserves, has a bad reputation. The language is well-known for its many quirks and pitfalls, as documented in various forms, ranging from online rants (informed to varied degrees) to books such as *Verilog and SystemVerilog Gotchas: 101 Common Coding Errors and How to Avoid Them* [55].

Yet, the many quirks of Verilog, we argue, is not a reason to give up on formalisation. This is because one must be careful to differentiate between, what we here call, semantic shortcomings and pragmatic shortcomings of Verilog. The two problems we presented in the previous section are examples of semantic shortcomings: inconsistencies and other types of problems that can be found in the standard rendering the actual semantics of the language unclear. In contrast, the problems that have given Verilog its bad reputation are pragmatic shortcomings of Verilog, by which we mean commonly misunderstood aspects of Verilog that arise from Verilog's, at times, antipedagogical design. Pragmatic shortcomings can be resolved by a careful reading of the standard, whereas semantic shortcomings can not. To be clear, this is not to say that the pragmatic shortcomings do not constitute a problem; it is only to say that they are not blockers for formalisation.

To clarify what we mean by pragmatic shortcomings, we now discuss some oft-cited grievances of Verilog that have *not* been problems in our formalisation work.

---

signments in the active region set. The same text occurs in the Verilog-2005 standard [8, p. 120]. This is not of great importance here since the reactive region set is out of scope for our work.

X values. Although difficult to use and make sense of intuitively,[10] the semantics of X values as defined in the Verilog standard does not, as far as we are aware, pose any particular problems to implement/formalise. VV hence supports X values without any problems known to us.

Z values. Although Z values are used for a different purpose than X values (recall, they are used to model tristate logic/multidriven nets), the semantics of Z values is similar to the semantics of X values and like X values pose no particular problem to implement/formalise.

Variables vs. nets. The difference between variables and nets seems to be a common source of confusion for beginner Verilog hardware designers. E.g., at the time of writing, both the second [10] and third [14] most upvoted questions tagged with the "verilog" tag at Stack Overflow ask about the difference between variables and nets (specifically, `reg` and `wire`, the two most common ways to declare variables and nets before SystemVerilog).[11,12] However, we find the standard quite clear on this point. The evaluation models of variables and nets are completely different: the value of a variable is decided by the last write to that variable, whereas the value of a net is decided by the resolution function of the net type of the net and the values of the net's drivers.

Initialisation of variables and nets. Perhaps confusingly, a variable declaration `logic a = b` declares a variable `a` with initial value `b` whereas a net declaration

[10] That X values are a common source of frustration is well-documented [43, 54, 56]. X values could potentially be understood as a type of symbolic execution. With that said, the kind of symbolic execution offered by Verilog does not fit neatly into the traditional soundness/completeness categories from the symbolic execution literature [18]. In short, we usually want to, somehow, relate our symbolic semantics (containing X-like values) to a concrete semantics (containing no X-like values): a symbolic semantics is sound if all behaviour modelled by the symbolic semantics is a subset of the behaviour of the concrete semantics, and a symbolic semantics is complete if all behaviour of the concrete semantics is modelled by the symbolic semantics. (Note that some authors use different terms for the same concepts.) Verilog is neither sound nor complete in the above sense since some constructs in Verilog overapproximate concrete behaviour (e.g., if one bit in the input to an arithmetic operator is X, then the entire result value must be X [16, p. 261]) and some constructs underapproximate behaviour (e.g., when branching on an X value in an if-statement, the false branch must always be taken [16, p. 299]).

[11] The most upvoted question is a now-closed question about best coding practices for hardware-description languages (such as, of course, Verilog).

[12] One can only speculate why this is. One reason might be that software languages have no analogous split between variables and nets. Another reason might be the confusingly named data type `reg`, which was used to declare variables before the synonymous data type `logic` was introduced in SystemVerilog. Confusingly, the `reg` data type only implies register semantics (that is, variable semantics), it does not imply that the data will necessarily be mapped to a storage element in synthesis. A declaration `reg foo` is the same as the declaration `var reg foo`, which in turn is the same as `var logic foo` since `reg` and `logic` are synonymous.

`wire a = b` declares a net `a` and a continuous assignment `assign a = b` for the net. This is because the value of a net is decided by its set of drivers, and a net therefore can not be given an initial value. Adding to the confusion, different version of the Verilog standard specify different semantics for variable initialisation. In older version of the standard, `logic a = b` is equivalent to a declaration `logic a` and a block `initial a = b`, whereas the most recent standard guarantees that initialisation happens before any other execution.

Blocking vs. nonblocking assignments. Another common source of problems in Verilog is the difference between blocking and nonblocking assignments; e.g., the top-50 Verilog questions on Stack Overflow features many questions on this topic [9, 11–13, 15]. Whereas blocking assignments can be explained by analogy to assignments in software languages, nonblocking assignments have no analogue in software languages. Anecdotally, nonblocking assignments are often explained as "happening in parallel", in terms of "delta cycles" (even though this is a VHDL term), or in terms of coding guidelines such that blocking assignments are for combinational logic (stateless logic) and nonblocking assignments are for sequential logic (stateful logic). In the Verilog standard, the differences between the two are made clear by how they interact with the Verilog event queue.

Delay constructs. Another type of construct one does not find in software languages are delay constructs (e.g., `#2 a = b` and `a = #2 b`). Delays belong to the nonsynthesisable subset of Verilog, but are important to support in VV to be able to build basic test benches inside VV to stimulate otherwise synthesisable modules. Continuous assignment delays are particularly interesting. Whereas delayed procedural assignments are simply added to future time slots, delayed continuous assignment in some cases cancel previously scheduled continuous assignment updates [16, p. 235–236]. In VHDL terminology, this is the difference between "inertial delay" and "transport delay". There are additional interesting delay constructs, such as net delays. There is clear room for improvements in how the standard specifies the semantics of both continuous assignment delays and net delays, but since delay constructs are not synthesisable, and therefore not within our immediate interests, we do not elaborate these issues in depth here.

7 The VV tool

We now discuss VV, the visual simulation tool we have built to visually debug the Verilog standard. VV implements and visualises the Verilog reference algorithm for simulation, as described in Sec. 4, including the algorithm's associated event queue. Here, we discuss the interface of VV and give some comments on how we have fleshed out the standard's frugal description of the reference algorithm.

Using VV. We return to Fig. 1, containing a screenshot of VV. The drop-down menu in the top-left of the interface contains a small collection of test modules (at the time of writing, 130 modules), which can be loaded into VV. We created

```
type rec event
 = EventContUpdate(int, value)
 | EventBlockUpdate(int, string, value)
 | EventNBA(string, value)
 | EventEvaluation(int)
 | EventDelayedEvaluation(int)
 | Events(array<event>)

type time_slot = {
 active: array<event>,
 inactive: array<event>,
 nba: array<event> }

type proc_running_state
 = ProcStateFinished
 | ProcStateRunning
 | ProcStateWaiting

type proc_state = {
 pc: int,
 state: proc_running_state }

type state = {
 // [...]
 env: Belt.Map.String.t<value>,
 cont_env: array<value>,
 proc_env: array<proc_state>,
 queue: array<(int, time_slot)>,
 monitor: option<(string,
                  /* ... */)> }
```

Fig. 4: The full `state` data type contains a few more fields not interesting enough to mention here and are therefore omitted in the presentation here (`// [...]`). Moreover, the actual `event` data type contains `event_id`s as well, but they are only used for React-based visualisation and do not affect any behaviour.

the test modules in the process of implementing and experimenting with VV, and the modules illustrate different aspects and corner-cases of the semantics of Verilog. The rest of the interface is as follows, from left-to-right: the first column of the interface contains the source code of the currently selected test module (the source code can also be manually edited); the second column contains the normalised result of parsing the module in the first column; the third column contains the current state of the simulation; and the fourth column contains the output of the run so far (from, e.g., `$display` and `$monitor` calls). The third column (which contains the current state of the simulation) contains in more detail, from top-to-bottom: the current simulation time (and simulation status), the current state of all variables and nets, the current state of all continuous assignment (i.e., the net drivers), the current state of all procedural process, the current event queue (which we describe in more detail below), and the currently installed monitor (if any).

Simulation in VV is driven by the user clicking the next simulation step to happen. Possible next steps for the simulation are marked in blue in the third column of the interface, e.g., an event in the event queue ready to execute or the simulation-time text when the current time slot is empty and the simulation is ready to progress to the next nonempty time slot. E.g., in the screenshot in Fig. 1, there is one blue-marked active event. Clicking the blue event causes VV to execute the event and update the event queue and other simulation state accordingly. After executing the clicked event, the simulator goes back into waiting for the next user decision. If needed, e.g., when there are multiple events to chose from, execution can be restarted by re-parsing the module.

Implementation of VV. We have implemented VV in ReScript [5], an OCaml dialect of JavaScript. We have used React [4] for the front-end of VV and Ohm [3] for parsing Verilog source code.

The structure of the simulation state is not made explicit in the reference algorithm, instead it is described by prose text spread throughout the standard. We have fleshed out this description as follows. The top-level state structure of VV, called `state`, containing, among other fields, the Verilog event queue, is defined as described in Fig. 4. The `state` data type contains the state of all variables and nets (`env`), the state of all drivers/continuous assignments (`cont_env`), the state of all procedural processes (`proc_env`), the state of the event queue (`queue`), and, optionally, a monitor (`monitor`).

The most important part of VV is its implementation of the Verilog event queue.[13] The event queue in the `state` data type is represented by a series of time slots of type `time_slot` indexed by the simulation time (an `int`) of the time slot. Each time slot (i.e., the type `time_slot`) consists of the regions `active`, `inactive`, and `nba`. In the subset of Verilog supported by VV, no field for the observed region is needed in the time slot data type since only ever monitor invocations are scheduled in the observed region and monitors are only ever scheduled for all future time slots, not for a specific time slot. (Monitors are instead represented using the `monitor` field in the `state` data type.)

The reference algorithm does not dictate the exact structure of events, instead, it only mentions that there are two types of events (see `execute_region` in Sec. 4): "update" events and "evaluation" events. In VV, the two categories are refined into the `event` data type containing six event types. The numbers of event types and the structure of each event types are not canonical, instead, the two are largely a consequence of how other components of the interpreter are implemented; i.e., other interpreters/semantics might end up with other refinements.

The six event types in VV are as follows. The event types `EventContUpdate` and `EventBlockUpdate`, respectively, represent a continuous assignment update and an update scheduled by a procedural blocking assignment, where the `int` in the event types is the index of the continuous assignment/procedural process. The two event types `EventEvaluation` and `EventDelayedEvaluation` both represent the start of execution of procedural processes, and are separated only because of a small (and not particularly interesting) edge case. The event types `EventNBA` and `Events` are used to represent nonblocking assignments: executing a nonblocking assignment schedules an `EventNBA` event in the relevant NBA region, and when the NBA events are later moved to the active region, following Meredith et al. [42], as discussed in Sec. 5, the order between them is preserved by grouping them inside an `Events` event. The `nba` field of `time_slot` will only ever contain `EventNBA` events and, similarly, `Events` events will only ever contain `EventNBA` events.

[13] Other implementation details of VV are not a particularly interesting; the rest of VV is a simple and straightforwardly implemented event-driven interpreter, which required no particular ingenuity to implement.

Regarding the two problems discussed in Sec. 5: awaiting an improved Verilog standard, VV for now only interleaves processes when they block, maintains the order of all NBA events, and does not allow NBA events to mix with any other events.

8 Related work

We discuss related work, including previous Verilog simulators and formalisations.

Verilog simulators. Multiple Verilog simulators exist today, such as the simulators shipped with large commercial hardware development environments such as Xilinx Vivado [61] and the open-source simulator Icarus Verilog [2] and Verilator [6]. There is little overlap between existing simulators and VV: existing simulators are batch tools designed for debugging real-world hardware designs, whereas VV is an interactive tool designed for debugging the Verilog standard. Simulation speed is the main driving force behind the design of existing simulators, whereas for VV performance largely unimportant. Even at the expense of performance, VV must be in an as simple and direct correspondence with (in other words, be as ontologically faithful to) the Verilog standard as possible. E.g., the event queue maintained by VV must be exactly as described by the Verilog standard rather than implemented for performance. Moreover, in VV, the full behaviour of Verilog must be exposed, e.g., all event schedules must be exposed. In this respect, VV has more in common with explicit-state model checking than traditional simulators, except that VV is driven by human rather than machine.

To our best knowledge, no existing simulator allows for its event queue to be inspected, whereas in VV it is the main functions of the tool.[14] Instead, analysis/debugging facilities of existing simulators are designed with the aim to help its users to find and understand bugs in Verilog designs (rather than bugs in the semantics of Verilog). Common debugging facilities in existing simulators include "printf debugging" (e.g., `$display` and `$monitor`) and waveform visualisation (see Fig. 6 in App. A for an example waveform output). Regardless, using or extending an existing simulator with support for visualising its event queue is not a sound approach to our work here. To force ourselves to scrutinise the various corner of the Verilog standard, we had no option but to build a new simulator from the ground up. Our work here is to debug the Verilog standard, and doing so is only possible by starting from zero and carefully and critically reading the standard.

Verilog formalisations. Meredith et al.'s [42] and Chen et al.'s [22] Verilog semantics are to date the most complete Verilog semantics available. Their semantics follows the standard closely, specifically, the Verilog-2005 standard [8]. Meredith et al. consider a similar subset of Verilog to us. Their semantics does, however, not include support for X and Z values. Moreover, Meredith et al. have misunderstood the difference between variables and nets, and as a result do not

[14] Not even the Verilog APIs (PLI/VPI) defined by the standard that allow "foreign language functions to access the internal data structures of a SystemVerilog simulation" [16, Ch. 36], do, as far as we are aware, allow for such inspection.

include proper support for nets in their semantics. Thereby, they (unwittingly) do not take into consideration net-specific features such as multidriven nets. They do, on the other hand, consider arrays, which we do not include. Chen et al. support a larger subset of Verilog since they are not, like us, foremost interested in the core concurrency semantics of synthesisable Verilog.

Both Meredith et al.'s and Chen et al.'s semantics are executable. Meredith et al.'s semantics is executable because it is implemented in the K framework, which allows for generation of, among other things, interpreters and model checkers. Chen et al. make their semantics executable by manually implementing it in Java. Another Verilog project that emphasises executability is Bowen et al. [21] (although instead referring to the same concept as their semantics being "animatable"). They formalise Verilog in the logic programming language Prolog, enabling printing traces of Verilog executions. Their supported subset of Verilog is minimal: e.g., they do not discuss features such as nonblocking assignments or nets.

Gordon's [28] early work on Verilog semantics is another project of note. The project covers many important Verilog features, such as nonblocking assignments and delays. Nets are included as well, but restricted to a single driver. The semantics presented is, however, informal (and, in places, nonstandard): the semantics is presented in prose form (and furthermore, hence, is not executable).

Other previous projects on the semantics of Verilog we are aware of do not follow the standard as closely as Meredith et al., Chen et al., and Gordon. We consider those projects to be either suggestions for alternative semantics for Verilog (rather than formalisations of the standard semantics) or semantics derived from the standard semantics designed to aid formal reasoning.

Verilog visualisations. We are not aware of previous work visualising Verilog's simulation semantics. For Verilog's synthesis semantics, Materzok [39] has developed DigitalJS, "a visual Verilog simulator for teaching". DigitalJS uses Yosys [60] as its front-end and visualises the synthesised output of Yosys. However, the visualisation does not explain the internals of the synthesis process – the synthesis tool is still a black box for the user (i.e., it is only its final output that is visualised). Similar visualisations, although not interactive, come bundled with e.g. Yosys itself and Vivado. (On the topic on visualising synthesis algorithms, although not directly related to Verilog, Nestor [45] has implemented CADAPPLETS (later ported to CADApps) for visualising a selection of synthesis algorithms.)

Other visualisations of programming languages and formal methods. Although not a common form of formalisation, visualisation is not unheard of in previous programming-language-theory-related work; e.g., in previous work we find: visualisations of programming languages and paradigms with steep learning curves, e.g., Homer's [31] work on visualising stack-based concatenative languages (which, as Homer remarks, has sometimes (humorously) been referred to as "write-only" languages), Greenberg and Blatt's [29] Shtepper which visualises the execution of shell scripts, Eisenstadt and Brayshaw's [26] Prolog visualisation work, and the lambda calculus visualisations collected by Pramod's [48]; visualisations of difficult corners of mainstream languages, e.g., the visualisation tool Loupe [49]

which helps users "understand how JavaScript's call stack/event loop/callback queue interact with each other" and Cooper's [24] visualisations of coroutine event loops in JavaScript; and visualisations of distributed systems [19].

In the formal-methods literature, one can find examples of visualisations of formal-method techniques (e.g., ribbon proofs for separation logic [59] or graphical proof assistants, designed for education, such as Holbert [46]) and examples of providing understandable presentations of proof states in program-verification tools (e.g., the proof-state visualisations in KeY [17] and Iris' proof mode [36,37]).

References

1. cocotb website. https://cocotb.org
2. Icarus Verilog website. http://iverilog.icarus.com
3. Ohm website. https://ohmjs.org
4. React website. https://react.dev
5. ReScript website. https://rescript-lang.org
6. Verilator website. https://veripool.org/verilator
7. IEEE standard hardware description language based on the Verilog hardware description language. IEEE Std 1364-1995 (1996). https://doi.org/10.1109/IEEESTD.1996.81542
8. IEEE standard for Verilog hardware description language. IEEE Std 1364-2005 (2006). https://doi.org/10.1109/IEEESTD.2006.99495
9. How to interpret blocking vs non blocking assignments in Verilog? https://stackoverflow.com/questions/4653284 (2011)
10. Using wire or reg with input or output in Verilog. https://stackoverflow.com/questions/5360508 (2011)
11. Verilog sequence of non blocking assignments. https://stackoverflow.com/questions/15718192 (2013)
12. <= assignment operator in Verilog. https://stackoverflow.com/questions/26727727 (2014)
13. Assigning values in Verilog: difference between assign, <= and =. https://stackoverflow.com/questions/27435703 (2014)
14. What is the difference between reg and wire in a Verilog module? https://stackoverflow.com/questions/33459048 (2015)
15. What is the difference between = and <= in Verilog? https://stackoverflow.com/questions/35435420 (2016)
16. IEEE standard for SystemVerilog–unified hardware design, specification, and verification language. IEEE Std 1800-2017 (2018). https://doi.org/10.1109/IEEESTD.2018.8299595
17. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M. (eds.): Deductive Software Verification – The KeY Book. Springer (2016). https://doi.org/10.1007/978-3-319-49812-6
18. Baldoni, R., Coppa, E., D'elia, D.C., Demetrescu, C., Finocchi, I.: A survey of symbolic execution techniques. ACM Computing Surveys **51**(3) (2018). https://doi.org/10.1145/3182657
19. Beschastnikh, I., Liu, P., Xing, A., Wang, P., Brun, Y., Ernst, M.D.: Visualizing distributed system executions. ACM Trans. Softw. Eng. Methodol. **29**(2) (2020). https://doi.org/10.1145/3375633

20. Bodin, M., Charguéraud, A., Filaretti, D., Gardner, P., Maffeis, S., Naudziuniene, D., Schmitt, A., Smith, G.: A trusted mechanised JavaScript specification. In: Symposium on Principles of Programming Languages (2014). https://doi.org/10.1145/2535838.2535876
21. Bowen, J.P., Jifeng, H., Qiwen, X.: An animatable operational semantics of the Verilog hardware description language. In: International Conference on Formal Engineering Methods (ICFEM) (2000). https://doi.org/10.1109/ICFEM.2000.873820
22. Chen, Q., Zhang, N., Wang, J., Tan, T., Xu, C., Ma, X., Li, Y.: The essence of Verilog: A tractable and tested operational semantics for Verilog. Proc. ACM Program. Lang. **7**(OOPSLA2) (2023). https://doi.org/10.1145/3622805
23. Chen, Q., Zhang, N., Wang, J., Tan, T., Xu, C., Ma, X., Li, Y.: The essence of Verilog: A tractable and tested operational semantics for Verilog (artifact) (2023). https://doi.org/10.5281/zenodo.10020331
24. Cooper, H.: Coroutine event loops in Javascript. https://x.st/javascript-coroutines (2012)
25. DeRemer, F., Kron, H.: Programming-in-the large versus programming-in-the-small. In: Proceedings of the International Conference on Reliable Software (1975). https://doi.org/10.1145/800027.808431
26. Eisenstadt, M., Brayshaw, M.: The transparent PROLOG machine (TPM): an execution model and graphical debugger for logic programming. The Journal of Logic Programming **5**(4) (1988). https://doi.org/10.1016/0743-1066(88)90001-5
27. Flake, P., Moorby, P., Golson, S., Salz, A., Davidmann, S.: Verilog HDL and its ancestors and descendants. Proceedings of the ACM on Programming Languages **4**(HOPL) (2020). https://doi.org/10.1145/3386337
28. Gordon, M.: The semantic challenge of Verilog HDL. In: Symposium on Logic in Computer Science (1995). https://doi.org/10.1109/LICS.1995.523251
29. Greenberg, M., Blatt, A.J.: Executable formal semantics for the POSIX shell. Proc. ACM Program. Lang. **4**(POPL) (2019). https://doi.org/10.1145/3371111
30. Haas, A., Rossberg, A., Schuff, D.L., Titzer, B.L., Holman, M., Gohman, D., Wagner, L., Zakai, A., Bastien, J.: Bringing the web up to speed with WebAssembly. In: Conference on Programming Language Design and Implementation (PLDI) (2017). https://doi.org/10.1145/3062341.3062363
31. Homer, M.: Interleaved 2D notation for concatenative programming. In: Workshop on Programming Abstractions and Interactive Notations, Tools, and Environments (PAINT) (2022). https://doi.org/10.1145/3563836.3568722
32. Huibiao, Z., Bowen, J.P., Jifeng, H.: Deriving operational semantics from denotational semantics for Verilog. In: Proceedings Eighth Asia-Pacific Software Engineering Conference (2001)
33. Jifeng, H., Huibiao, Z.: Formalising Verilog. In: International Conference on Electronics, Circuits and Systems (ICECS) (2000). https://doi.org/10.1109/ICECS.2000.911568
34. Jifeng, H., Qiwen, X.: An operational semantics of a simulator algorithm. Tech. Rep. 204, International Institute for Software Technology, United Nations University (2000)
35. Klimis, V., Clark, J., Baker, A., Neto, D., Wickerson, J., Donaldson, A.F.: Taking back control in an intermediate representation for gpu computing. Proc. ACM Program. Lang. **7**(POPL) (2023). https://doi.org/10.1145/3571253
36. Krebbers, R., Jourdan, J.H., Jung, R., Tassarotti, J., Kaiser, J.O., Timany, A., Charguéraud, A., Dreyer, D.: MoSeL: A general, extensible modal framework for

interactive proofs in separation logic. Proceedings of the ACM on Programming Languages **2**(ICFP) (2018). https://doi.org/10.1145/3236772
37. Krebbers, R., Timany, A., Birkedal, L.: Interactive proofs in higher-order concurrent separation logic. In: Symposium on Principles of Programming Languages (POPL) (2017). https://doi.org/10.1145/3009837.3009855
38. Lamport, L.: How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Transactions on Computers **C-28**(9) (1979). https://doi.org/10.1109/TC.1979.1675439
39. Materzok, M.: DigitalJS: A visual Verilog simulator for teaching. In: Proceedings of the 8th Computer Science Education Research Conference (2019). https://doi.org/10.1145/3375258.3375272
40. Memarian, K., Gomes, V.B.F., Davis, B., Kell, S., Richardson, A., Watson, R.N.M., Sewell, P.: Exploring C semantics and pointer provenance. Proceedings of the ACM on Programming Languages **3**(POPL) (2019). https://doi.org/10.1145/3290380
41. Memarian, K., Matthiesen, J., Lingard, J., Nienhuis, K., Chisnall, D., Watson, R.N.M., Sewell, P.: Into the depths of C: Elaborating the de facto standards. In: Conference on Programming Language Design and Implementation (PLDI) (2016). https://doi.org/10.1145/2908080.2908081
42. Meredith, P., Katelman, M., Meseguer, J., Roşu, G.: A formal executable semantics of Verilog. In: International Conference on Formal Methods and Models for Codesign (MEMOCODE) (2010). https://doi.org/10.1109/MEMCOD.2010.5558634
43. Mills, D.: Being assertive with your X. In: Synopsys Users Group Conference (SNUG) (2004)
44. Mills, D.: Yet another latch and gotchas paper. In: Synopsys Users Group Conference (SNUG) (2012)
45. Nestor, J.A.: Experience with the CADAPPLETS project. IEEE Transactions on Education **51**(3) (2008). https://doi.org/10.1109/TE.2008.919650
46. O'Connor, L., Amjad, R.: Holbert: Reading, writing, proving and learning in the browser. In: Human Aspects of Types and Reasoning Assistants (HATRA) (2022). https://doi.org/10.48550/ARXIV.2210.11411
47. Pace, G.J.: Hardware Design Based on Verilog HDL. Ph.D. thesis, Oxford University (1998)
48. Pramod, P.: Lambda calculus visualizations. https://github.com/prathyvsh/lambda-calculus-visualizations
49. Roberts, P.: Loupe web page. http://latentflip.com/loupe (2014)
50. Roșu, G., Șerbănuță, T.F.: An overview of the K semantic framework. The Journal of Logic and Algebraic Programming **79**(6) (2010). https://doi.org/10.1016/j.jlap.2010.03.012
51. Schneider, G., Qiwen, X.: Towards an operational semantics of Verilog. Tech. Rep. 147, International Institute for Software Technology, United Nations University (1998)
52. Schneider, G., Xu, Q.: Towards a formal semantics of Verilog using duration calculus. In: Formal Techniques in Real-Time and Fault-Tolerant Systems (1998)
53. Stewart, D.: A Uniform Semantics for Verilog and VHDL Suitable for Both Simulation and Verification. Ph.D. thesis, University of Cambridge (2002)
54. Sutherland, S.: I'm still in love with my X! In: Design and Verification Conference (DVCon) (2013)
55. Sutherland, S., Mills, D.: Verilog and SystemVerilog Gotchas: 101 Common Coding Errors and How to Avoid Them. Springer (2007). https://doi.org/10.1007/978-0-387-71715-9

56. Turpin, M.: The dangers of living with an X. In: Synopsys Users Group Conference (SNUG) (2003)
57. Watt, C.: Mechanising and verifying the WebAssembly specification. In: International Conference on Certified Programs and Proofs (2018). https://doi.org/10.1145/3167082
58. Watt, C., Rao, X., Pichon-Pharabod, J., Bodin, M., Gardner, P.: Two mechanisations of WebAssembly 1.0. In: International Symposium on Formal Methods (FM) (2021). https://doi.org/10.1007/978-3-030-90870-6_4
59. Wickerson, J., Dodds, M., Parkinson, M.: Ribbon proofs for separation logic. In: European Symposium on Programming (ESOP) (2013). https://doi.org/10.1007/978-3-642-37036-6_12
60. Wolf, C.: Yosys open synthesis suite. https://yosyshq.net/yosys
61. Xilinx: Vivado Design Suite User Guide: Getting Started (UG910, v2022.2) (2022)
62. Zhu, H., He, J., Bowen, J.P.: From algebraic semantics to denotational semantics for Verilog. In: International Conference on Engineering of Complex Computer Systems (ICECCS) (2006). https://doi.org/10.1109/ICECCS.2006.1690363
63. Zhu, H., He, J., Bowen, J.P.: From operational semantics to denotational semantics for Verilog. In: Correct Hardware Design and Verification Methods (CHARME) (2001)

A Example circuit and test bench

Fig. 5 contains a small hardware-design module `circuit`, which is synthesisable, and a test bench module `circuit_tb`, which is not nonsynthesisable. The test bench `circuit_tb` stimulates and probes the hardware description `circuit` much like a real physical test bench would do. The test bench module is not synthesisable because of how timing and event controls are used to define the module: the clock is modelled using a delayed assignment and the input stimuli process contains multiple event controls.

```
module circuit(
 input logic clk,
 input logic inp1,
 input logic inp2,
 output logic out);

// Model of register with
// xor of inp1 and inp2
// as input
always_ff @(posedge clk)
 out <= inp1 ^ inp2;

endmodule
```

```
module circuit_tb;

logic clk = 0, inp1, inp2, out;

// Instantiate the design, connecting
// up inputs and outputs by name
circuit circuit(.*);

// Behavioural model of a clock
always #1 clk = !clk;

// Install a monitor that will print
// values (when changed) at the end
// of each time slot
initial begin
 $monitor("time = %0d --> ", $time,
          "inp1 = %b, inp2 = %b, out = %b",
          inp1, inp2, out);
end

// Input stimuli for the design
initial begin
 @(posedge clk) inp1 <= 1; inp2 <= 0;
 @(posedge clk) inp2 <= 1;
 @(posedge clk) @(negedge clk) $finish;
end

endmodule
```

Fig. 5: Example hardware design `circuit` and test bench `circuit_tb`

Running the circuit in its test bench using the Verilog simulator Icarus [2], we receive the following output (the `-g2012` flag enables SystemVerilog support):

```
> iverilog -g2012 circuit.sv circuit_tb.sv
> ./a.out
time = 0 --> inp1 = x, inp2 = x, out = x
```

```
time = 1 --> inp1 = 1, inp2 = 0, out = x
time = 3 --> inp1 = 1, inp2 = 1, out = 1
time = 5 --> inp1 = 1, inp2 = 1, out = 0
circuit_tb.sv:22: $finish called at 6 (1s)
```

Fig. 6 contains a waveform visualisation of the same run.

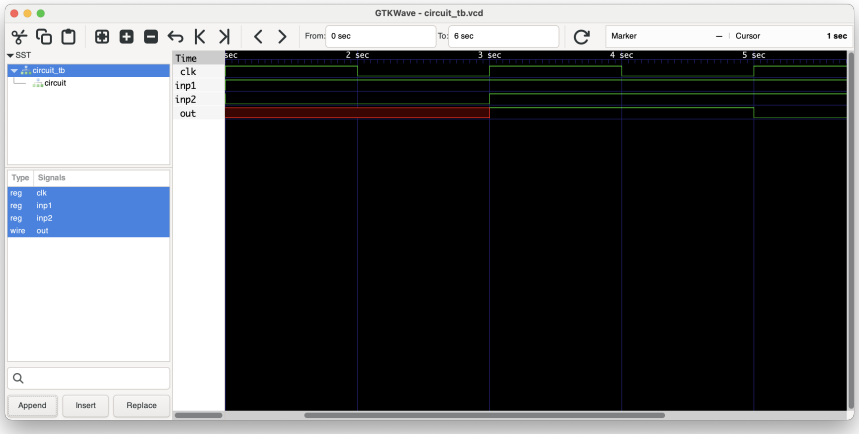

Fig. 6: A screenshot of GTKWave with a run of `cirucit_tb`. The waveform display shows the states of data objects over time.

B First identified problem

Since we will be referring to the Verilog semantics of Meredith et al. [42] and Chen et al. [22] repeatedly in this and the following appendix, we here and in the following appendix refer to the two semantics as K-Verilog and λ-Verilog, respectively. Both semantics have state-explorer tools, allowing us to explore all reachable behaviour of the semantics. λ-Verilog can also be run as an interpreter, where we can supply a "seed" to control which schedule is used during execution. For modules that we run in K-Verilog and λ-Verilog, we must use `always @(*)` instead of `always_comb` because K-Verilog and λ-Verilog are formalisations of Verilog, not SystemVerilog. Here, the only difference between `always @(*)` and `always_comb` is that the latter automatically runs once in the first time slot.

First, to show that the interleaving we discuss in the main text for the module `interleave` from Fig. 3 is possible in both K-Verilog and λ-Verilog, consider the following variant of the module:

```
module interleave_observable;

// We use an array here since
// K-Verilog does not support
// X values
reg[1:0] a, b;

always @(*)
 b = a;

initial begin
 a = 1;
 a = 2;
end

initial $monitor("a = %b, b = %b", a, b);

endmodule
```

Running the λ-Verilog interpreter with seed 38 gives the following output:

```
> lv -ci interleave_observable.v --seed=38 -o tmp.lv
a = 10, b = 01
```

The state-space explorer of K-Verilog confirms that the same outcome is possible in the K-Verilog semantics.

We also consider the same problem in a more realistically structured setup, with a separate hardware model and test bench:

```
module interleaving(
 input wire[1:0] a,
 output reg[1:0] b,
 output reg[1:0] c);
```

```
// First combinational logic block,
// assigns b twice
always @(*) begin
 // First assignment of b
 b = 0;

 // Second assignment of b
 b = a + 1;
end

// Second combinational logic block,
// depends on b
always @(*)
 c = b;

endmodule

module interleaving_tb;

reg[1:0] a;

wire[1:0] b, c;

initial begin
 #1 a <= 1;
 #1 a <= 2;
 #5 $finish;
end

initial $monitor("a = %b, b = %b, c = %b", a, b, c);

interleaving interleaving(.a(a), .b(b), .c(c));

endmodule
```

To some extent, this more realistic example is still artificial because the first combinational block only contains two assignments to the same variable, making the first assignment redundant. However, this type of code occur in real-world code. E.g., the following code is example 4.5c "case with defaults listed before case statement" from Mills [44] which illustrates a coding-style sometimes used to avoid inferring latches:

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

Note that both `out1` and `out2` are assigned multiple times in the block.

Returning back to `interleaving` and `interleaving_tb`: we have run the code in the five different Verilog simulators available at `https://edaplayground.com` and they all output the following:

```
a = xx, b = xx, c = xx
a = 01, b = 10, c = 10
a = 10, b = 11, c = 11
```

In contrast, when the modules are adapted for K-Verilog and λ-Verilog, states where `b` $\neq$ `c` are reachable in both semantics, which, as discussed in the main text, is not compatible with the semantics of hardware.

Turning the level of realism further up, we now consider one of the test cases from Chen et al.'s `scripts/data-race-cases.list` file [23], which blacklists 99 test cases from the Icarus test suite that do not work correctly under their semantics, specifically `talu.v`. The unmodified test case is as follows:

```
/* Copyright (C) 1999 Stephen G. Tell
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2, or (at your option)
 * any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this software; see the file COPYING.  If not, write to
 * the Free Software Foundation, Inc., 59 Temple Place, Suite 330,
 * Boston, MA 02111-1307 USA
 */

/* talu - a verilog test,
 * illustrating problems I had in fragments of an ALU from an 8-bit micro
 */

module talu;
   reg error;

   reg [7:0] a;
   reg [7:0] b;
   reg cin;
   reg [1:0] op;

   wire cout;
   wire [7:0] aluout;
```

```
alu alu_m(a, b, cin, op, aluout, cout);

initial begin
    error = 0;

    // add
    op='b00; cin='b0; a='h0; b='h0;
    #2 if({cout, aluout} != 9'h000) begin
        $display($time, " FAILED %b  %b %h %h  %b %h",
                 op, cin, a, b, cout, aluout);
        error = 1;
    end

    // add1
    op='b01; cin='b0; a='h01; b='h01;
    #2 if({cout, aluout} != 9'h103) begin
        $display($time, " FAILED %b  %b %h %h  %b %h",
                 op, cin, a, b, cout, aluout);
        error = 1;
    end

    // and
    op='b10; cin='b0; a='h16; b='h0F;
    #2 if({cout, aluout} != 9'h006) begin
        $display($time, " FAILED %b  %b %h %h  %b %h",
                 op, cin, a, b, cout, aluout);
        error = 1;
    end
    op='b10; cin='b0; a='h28; b='hF7;
    #2 if({cout, aluout} != 9'h020) begin
        $display($time, " FAILED %b  %b %h %h  %b %h",
                 op, cin, a, b, cout, aluout);
        error = 1;
    end

    // genbit
    op='b11; cin='b0; a='h00; b='h03;
    #2 if({cout, aluout} != 9'h008) begin
        $display($time, " FAILED %b  %b %h %h  %b %h",
                 op, cin, a, b, cout, aluout);
        error = 1;
    end
    op='b11; cin='b0; a='h00; b='h00;
    #2 if({cout, aluout} != 9'h001) begin
        $display($time, " FAILED %b  %b %h %h  %b %h",
                 op, cin, a, b, cout, aluout);
        error = 1;
    end

    /* tests are incomplete - doesn't compile yet on ivl */
```

```
        if(error == 0)
            $display("PASSED");

        $finish;
    end

endmodule

/* fragments of an ALU from an 8-bit micro
 */

module alu(Aval, Bval, cin, op, ALUout, cout);
    input [7:0]  Aval;
    input [7:0]  Bval;
    input        cin;
    input [1:0]  op;
    output       cout;
    output [7:0] ALUout;

    reg          cout;
    reg [7:0]    ALUout;

    always @(Aval or Bval or cin or op) begin
       case(op)
         2'b00 : {cout, ALUout} = Aval + Bval;
         2'b10 : {cout, ALUout} = {1'b0, Aval & Bval};

// C++ compilation troubles with both of these:
         2'b01 : {cout, ALUout} = 9'h100 ^ (Aval + Bval + 9'h001);
         2'b11 : {cout, ALUout} = {1'b0, 8'b1 << Bval};

//         2'b01 : {cout, ALUout} = 9'h000;
//         2'b11 : {cout, ALUout} = 9'h000;
       endcase
    end // always @ (Aval or Bval or cin or op)

endmodule
```

To make this test case pass under their semantics, Chen et al. add the following synchronisation to the test bench:

```
    initial begin
        error = 0;

        // add
        op='b00;
        #0; // wjp: synchronize simultaneous signal change
        cin='b0;
        #0; // wjp: synchronize simultaneous signal change
```

```
        a='h0;
        #0; // wjp: synchronize simultaneous signal change
        b='h0;
        #0; // wjp: synchronize simultaneous signal change
        #2 if({cout, aluout} != 9'h000) begin
            $display($time, " FAILED %b  %b %h %h  %b %h",
                     op, cin, a, b, cout, aluout);
            error = 1;
        end

        // add1
        op='b01;
        #0; // wjp: synchronize simultaneous signal change
        cin='b0;
        #0; // wjp: synchronize simultaneous signal change
        a='h01;
        #0; // wjp: synchronize simultaneous signal change
        b='h01;
        #0; // wjp: synchronize simultaneous signal change
        #2 if({cout, aluout} != 9'h103) begin
            $display($time, " FAILED %b  %b %h %h  %b %h",
                     op, cin, a, b, cout, aluout);
            error = 1;
        end

        // etc.

    end
```

That is, this module illustrates a problem with unrestricted interleavings different from the double-write problem discussed up till this point. The hardware model depends on multiple data objects, and when the hardware model can preempt at any point, it can miss any number of writes to these data objects. Therefore, Chen et al. have added delays between the writes to ensure that the hardware model runs to completion before the next update is applied.

To conclude, for this paper, we cannot do more than to point out the incompatibility between the interleaving semantics the standard suggest and the interleaving semantics assumed by current Verilog practice; this is because actually addressing the incompatibility will require discussions and agreement between the various stakeholders of Verilog. With that said, below we have summarised the interleaving choices made by a few existing Verilog tools and other relevant sources, which we hope will help inform further discussions on the topic:

The Verilog standard From the very first Verilog standard [7] to the latest Verilog standard [16], the standards specify that procedural processes are allowed to preempt at any point during execution.

Mathematical formalisations of Verilog As shown above, the semantics of Meredith et al. and Chen et al. both allow arbitrary preemption.

Open-source simulators Open-source simulators, by definition, allow us to inspect their source code and we can therefore easily know what implementation choices that have been made in those tools. The two main open-source simulators are Icarus [2] and Verilator [6]. However, the semantics that Verilator implements is closer to Verilog's synthesis semantics than Verilog's simulation semantics, and we therefore only consider Icarus here. Icarus does not preempt procedural processes, however, for simple enough continuous assignments, such continuous assignments are updated immediately, which is a limited form of preemption. The following example illustrates this:

```
module ex1;

logic i, o1, o2;

// Simple continuous assignment
assign o1 = i;

// Nonsimple continuous assignment
assign o2 = i + 1;

initial begin
 $display("i = %b, o1 = %b, o2 = %b", i, o1, o2);
 i = 1;
 $display("i = %b, o1 = %b, o2 = %b", i, o1, o2);
end

endmodule
```

When run in Icarus, we get the following output:

```
i = x, o1 = x, o2 = x
i = 1, o1 = 1, o2 = x
```

Closed-source simulators For closed-source simulators, we cannot get a definitive answer to how they preempt and interleave processes. We have run some small-scale systematic experiments with the close-source Verilog simulators available at `https://edaplayground.com`, and it *appears* to be the case that processes are not interleaved except for corner cases. Below we present two examples of behaviour equivalent to process preemption that can be observed in two existing simulators.

The first example is for the simulator Aldec Riviera Pro 2022.04:

```
module ex1;

logic i, o;

buf (o, i);

initial begin
 $display("i = %b, o = %b", i, o);
 i = 1;
```

```
 $display("i = %b, o = %b", i, o);
end

endmodule
```

The simulator outputs:

```
i = x, o = x
i = 1, o = 1
```

The second example is for the simulator Synopsys VCS 2021.09. The following module shows that the simulator sometimes interleaves continuous assignments and `initial` blocks:

```
module ex2;

logic i, o1, o2;

initial begin
 $display("i = %b, o1 = %b, o2 = %b", i, o1, o2);
 i = 1;
 $display("i = %b, o1 = %b, o2 = %b", i, o1, o2);
end

assign o1 = i;
assign o2 = i + 1;

endmodule
```

For the above module, the simulator outputs:

```
i = x, o1 = x, o2 = x
i = 1, o1 = 1, o2 = 0
```

However, note that for the same module with the `initial` block replaced by an `always_comb` block, the behaviour changes:

```
module ex3;

logic i, o1, o2;

always_comb begin
 $display("i = %b, o1 = %b, o2 = %b", i, o1, o2);
 i = 1;
 $display("i = %b, o1 = %b, o2 = %b", i, o1, o2);
end

assign o1 = i;
assign o2 = i + 1;

endmodule
```

We now instead get the output:

```
i = x, o1 = x, o2 = x
i = 1, o1 = x, o2 = x
i = 1, o1 = 1, o2 = 0
i = 1, o1 = 1, o2 = 0
```

I.e., in this case, the simulator does not interleave the two.

C Second identified problem

Both K-Verilog and λ-Verilog allow NBA events to mix with other events. To exemplify, consider the following version of the module `nbinterleave1` from Fig. 3:

```
module nbinterleave1;

reg[1:0] a;

initial begin
 a <= 1;
 a <= 2;
end

always @(*)
 $display("a = %b", a);

endmodule
```

The K-Verilog state-space explorer reports the following as possible outputs:

```
a = 1
```

and

```
a = 10
```

and

```
a = 1
a = 10
```

λ-Verilog also has interleavings of the NBA events and the display process:

```
> lv -ci nbinterleave1.v --seed=12 -o tmp.lv
a = 01
a = 10
```

Because both K-Verilog and λ-Verilog mix NBA events with other events, we can easily observe that none of them reorder NBA events for different variables. To exemplify, consider the following Verilog module:

```
module nbinterleave2;

reg a, b;

initial begin
 a <= 1;
 b <= 1;
end

always @(*)
```

```
 $display("a = %b, b = %b", a, b);

endmodule
```

For K-Verilog, the following three output are possible:

```
a = 1, b = 0
```

and

```
a = 1, b = 1
```

and

```
a = 1, b = 0
a = 1, b = 1
```

The state-space explorer of λ-Verilog reports:

```
> lv -cx nbinterleave2.v -o tmp.lv | grep -v '^Heap'
a = 1, b = x
a = 1, b = x
a = 1, b = 1
a = 1, b = 1
a = 1, b = 1
a = 1, b = 1
```

That is, for both semantics, `b` is never assigned before `a`.