A Visual Formalisation of Verilog

Andreas Lööw, Imperial College London, UK

Abstract

Infamously, Verilog’s so-called simulation semantics, defined by the Verilog standard, is difficult to formally pin down. In this paper, as an intermediate step towards a mathematical formalisation of Verilog’s simulation semantics, we explore an, what we believe to be, underappreciated form of semantics formalisation: namely, visual formalisation.

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 programming language or HDL is required. For a language with a mathematical formalisation, applications of formal reasoning include the verification of programs/hardware designs implemented in the language and the verification of tools for the language, such as compilers/synthesis tools and analysis tools (e.g., model checkers).

Despite Verilog being the most popular HDL [37], no definitive mathematical formalisation of the language exists today. In this paper, we suggest that a visual formalisation of Verilog is a useful intermediate step towards a definitive mathematical formalisation of Verilog and contribute a new visual formalisation of Verilog in the form of a visual simulation tool VV (“Verilog visualiser”). The tool is web based, and the source code and a live demo of the tool are available at https://github.com/andreasloow/vv.

Problem: Verilog. The semantics of Verilog, its so-called simulation semantics, is defined by the Verilog standard, IEEE 1800-2017 [17] (which, formally, is the SystemVerilog standard, since Verilog has been merged into SystemVerilog).1,2 Verilog’s simulation semantics is concurrent and event driven: the semantics 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.

As is common for language standards, the Verilog standard is written in prose form, that is, the standard is not formal. The standard is known to be difficult to read and understand, rendering formalisation of the standard difficult. Previous work on understanding and formalising the simulation semantics of Verilog exist but no previous work provides a definitive formalisation. The most important point of comparison is Meredith et al.’s [49] Verilog semantics since it is the current most complete and accurate formalisation. As we explain in this paper, Meredith et al. appear to have misunderstood important parts of the standard. Other previous formalisation work either covers a smaller subset of Verilog than Meredith et al. and/or suggest alternative semantics for Verilog rather than formalise Verilog’s simulation semantics. No previous formalisation of Verilog’s simulation semantics has been visual. (We discuss related work in more detail in Sec. 7 and 8.)

Approach: Visual formalisation. We believe that the visual formalisation of Verilog we contribute in this paper has the potential to serve as a useful intermediate step towards a definitive mathematical formalisation of Verilog.

This is because we hypothesise that the reason the literature has yet to see a definitive mathematical formalisation of Verilog is the result of a communal split between the hardware-design community and the programming-language-theory community: the hardware-design community has the know-how of Verilog but not the know-how of developing mathematical formalisations of languages whereas the programming-language-theory community has the opposite problem.

Our visual formalisation of Verilog, we believe, has the potential to enable the two communities to combine their expertise. We believe this because, in contrast to a mathematical formalisation, no specialist mathematical knowledge is required to understand our visual formalisation, hence, our visual formalisation enables the hardware-design community to judge if our formalisation is consistent with their everyday intuitions about Verilog. Simultaneously, in contrast to the Verilog prose standard, extensive knowledge of Verilog is not required to understand our visual formalisation, hence, our visual formalisation provides a first starting point for a mathematical formalisation of Verilog for the programming-language-theory community.

Contribution: A visual formalisation of Verilog. To address the above, we contribute the new visual Verilog simu-
lation tool VV. We also highlight problems we have found in the Verilog standard in the process of developing VV.

VV provides a visual formalisation of Verilog’s simulation semantics by visualising 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 covers the core of synthesisable Verilog, that is, the subset of Verilog used to implement hardware designs, and a small select subset of nonsynthesisable Verilog, such that stimuli can be provided to hardware designs implemented in the synthesisable subset of Verilog supported by VV. We target synthesisable Verilog because it is the part of Verilog we see as most important for future mathematical formalisation since it is the part of Verilog required for formally reasoning about both Verilog hardware designs and tools such as synthesis tools and analysis tools for Verilog hardware designs.

2. Formalisations and visual formalisations

We now elaborate on what relationship we see between mathematical formalisations and visual formalisations of languages, specifically, how visual formalisations can help the development of mathematical formalisations, in particular for Verilog.

Developing 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 [30] work on JavaScript, Memarian et al.’s [47, 48] work on C, and Klimis et al.’s [42] work on GPU computing, and (less commonly) co-developing standards along with mathematical formalisations, e.g., the work on WebAssembly [40, 59, 60].

When working with an existing prose standard (as in the case of Verilog), the evaluation criterion of a mathematical formalisation of the standard is relatively straightforward: a formalisation is a successful formalisation if it in some sense “captures” and “corresponds” to the standard. Bodin et al. [30], in their work on formalising JavaScript, suggests the term “eyeball closeness” for this evaluation criterion. Eyeball closeness captures the idea that a formalisation and the standard it is an interpretation of should be in such close and direct correspondence that if they are put next to each other, a quick eyeballing of the two should allow us to conclude with some confidence that they are in correspondence. In other words, the two must build on the same fundamental concepts and correspond to each other ontologically in an as direct sense as possible.

At first glance, developing a mathematical formalisation of a prose standard might appear to be a relatively straightforward activity as well: first, (1) carefully read the prose standard, and then, (2) carefully write down what was read in the form of a mathematical formalisation.

This two-step summary of the formalisation process, however, obscures the interpretative work that is often required in practice to develop a formalisation. In practice, because of the inherent ambiguity and imprecision of prose text, creative use of alternative sources supporting the required interpretative work of the standard is required to fill in the gaps in the standard or addressing errors and outright contradictions of the standard. We must rely on a complex of sources, together helping us to tease out the fundamental concepts of the standard such that we can write them down carefully and clearly in the form of a formalisation for which we can, with some confidence, claim to be in correspondence with the standard. For example, Bodin et al., in their work on formalising JavaScript, cites the JavaScript prose standard, browser implementations of JavaScript, discussion groups such as es-discuss, and the official ECMA test suite test262 as their complex of sources. Memarian et al. [47, 48], in their work on formalising C, cites an even larger 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.

Visual formalisations. We see visual formalisations as (underappreciated and) useful members of languages’ complex of sources that inform the design of mathematical formalisations of the languages. In particular, whereas many traditional sources, such as test suites, allow for empirical evaluation of formalisations, a visual formalisation allows for analytical evaluation, where formalisation users can essentially play around with test cases to find problems with the formalisation or the language itself. In fact, this type of ad hoc exploration, using VV, is how we have found the problems in the semantics of Verilog we describe later in the paper. We see analytical evaluation as a complement to empirical evaluation, not a replacement: analytical evaluation will find problems empirical evaluation will not and vice versa.

For Verilog specifically, we see visual formalisation to be a particular good fit because of the communal split described in the introduction; that is, the communal split between the between the hardware-design community and the programming-language-theory community. Because of the communal split, we see a need for a source in Verilog’s complex of sources that is a reasonable trade-off between the two conflicting requirements of accessibility and precision, i.e., that that is understandable by both communities and can simultaneously feed into a definitive mathematical formalisation of Verilog.

3. The subset of Verilog supported by VV

In this section, we introduce and discuss the subset of Verilog that is supported by VV. Here, we only give informal comments on the semantics of the subset. In Sec. 4, we discuss how the Verilog standard defines the semantics of Verilog, in particular, the semantics of the subset of Verilog supported by VV. In Sec. 5, we discuss how VV implements and visualises the semantics of Verilog.
3.1. The supported subset

Fig. 1 describes the grammar of the subset of Verilog supported by VV, which we now discuss in bottom-up order.

Values. Verilog bits can take on four different values [17, p. 83]. All four are included in VV, see $v$ (in Fig. 1). The value $x$ is used for a multitude of purposes, often representing something like “unknown value”, “invalid value”, “error”, or similar. The value $z$ represents a high-impedance state and is used to model tristate logic, as illustrated below.

Data objects. A data object in Verilog is a “named entity that has a data value and a data type associated with it” [17, p. 83]. All four are included in VV, see $mi$. 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 VV, a set of continuous assignments, see below. More precisely, 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 in VV: wire, wor, and wand. E.g., for a wire net, all drivers with value $z$ 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. Recall that Verilog is a concurrent language. Verilog is based on processes, and there are two types of processes included in the subset of Verilog supported by VV: procedural processes and continuous assignments. See, again, $mi$.

Procedural processes arise from always blocks, initial blocks, etc., which induce software-like processes (with a program counter, internal state, etc.). 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-ware audiences, such as the various timing- and event-control constructs used to coordinate process execution. Which we explain these potentially less familiar faces 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.

The second type of process arises 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. 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 VV, we consider one module at a time.
3.2. Subset selection criteria

Verilog is a large language, the Verilog standard weights in at 1315 pages. The subset of Verilog we have carved out here might appear worryingly small. Indeed, the grammar of our subset fits into the small figure Fig. 1, whereas the grammar of full Verilog as presented in appendix A of the Verilog standard requires 45 full pages.

We now therefore discuss how and why we have selected the subset of Verilog supported by VV.

Restriction 1: synthesisable Verilog. Recall that our ultimate aim is a mathematical formalisation of Verilog. The main usage of such a formalisation is to reason formally about Verilog hardware designs and tools manipulating and analysing Verilog hardware designs, for which only the subset of Verilog (a) Example test bench code (nonsynthesisable Verilog)

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

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

(b) Example hardware design (synthesisable Verilog)

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

// Model of a clock
always @(posedge clk)
    out <= inp1 ^ inp2;
endmodule
```

(c) Output from running the test bench using the Verilog simulator Icarus

```bash
> iverilog -g2012 circuit_tb.sv circuit.sv
> ./a.out

> time = 4, clk = 0 --> inp1 = 1, inp2 = 1, out = 0

Figure 2: Example test bench circuit_tb and hardware design circuit
```

Even outside the domain of formal reasoning, various approaches to hardware development replace Verilog test benches with various other infrastructures. E.g., cocotb [2] enables implementing test benches in Python instead of (nonsynthesisable) Verilog.

5Strictly speaking, what part of Verilog is considered synthesizable or not depends on one’s synthesis tool and technology that is targeted. E.g., some target technologies support specifying the initial values of registers whereas others do not. However, for our purposes here, these various flavours of “synthesizable Verilog” are similar enough that it is sufficiently precise to talk about “synthesizable Verilog” without specifying what synthesis tool or target technology we have in mind.

6Even outside the domain of formal reasoning, various approaches to hardware development replace Verilog test benches with various other infrastructures. E.g., cocotb [2] enables implementing test benches in Python instead of (nonsynthesisable) Verilog.

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

// Model of a clock
always @(posedge clk)
    out <= inp1 ^ inp2;
endmodule
```
Lastly and relatedly, Verilog contains many constructs for supporting programming-in-the-large [35] (in contrast to programming-in-the-small), such as 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 the constructs do not complicated the event queue further. One borderline construct is arrays, which we do not include. Not including arrays allows us to avoid a long series of minor (non-concurrency) problems we are not directly interested in for the moment, such as the implicit resizing semantics of Verilog.7

4. The simulation semantics of Verilog

The Verilog standard defines the simulation semantics of Verilog by giving pseudocode for a “reference algorithm for simulation” [17, Sec. 4.5] in combination with prose text sprinkled throughout the standard. In this section, we recapitulate the reference algorithm as presented in the standard. In the next section, we describe how we have implemented the algorithm in VV and highlight two problems we have found in the reference algorithm while implementing VV.

To implement a Verilog simulator is to implement the Verilog reference algorithm for simulation. 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 Verilog 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 time slot, or “simulation time”. Each time slot is split into “regions”. The following regions are relevant for the subset of Verilog supported by VV: active, inactive, NBA (“nonblocking assignment”), and observed.8 With some important exceptions, nondelayed events are scheduled in the current time slot’s active region, zero-delayed events are scheduled in the current region’s inactive region, and nonzero-delayed events are scheduled in a future time slot’s active region. Process execution, including the execution of blocking assignments, happen in the active region. Nonblocking assignments differ from this general pattern in that they are scheduled in the NBA region. This enables nonblocking assignments to be used for communication between processes since they do not race with events scheduled in the active region, such as process execution. The observed region is used by the smonitor system task, to print values at end of time slots (like in the test bench in Fig. 2).

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;
}

Regions, in turn, are executed by the pseudocode function execute_region:

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 possibly schedule further events for execution;
}
}

Executing a region is (like to the reference algorithm as a whole) driven by processes creating events – so-called update events – and reacting to events – so-called evaluation events.

5. The VV tool

Our new visual simulation tool VV implements and visualises the Verilog reference algorithm for simulation, as described in the previous section, including the algorithm’s associated event queue. In this section, we discuss VV and discuss two problems with the reference algorithm, that is, the semantics of Verilog, we have found while implementing VV. In the next section, we give further commentary on the reference algorithm and semantics of Verilog based on our experiments with VV.

Using VV. Fig. 3 contains a screenshot of VV. The only requirement to run VV is a web browser. VV is bundled with a small collection of test modules (at the time of writing, 130 modules), which can be loaded into VV using the drop-down
menu to the top-left of the interface. We created 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. We refer to some of the modules in this and the next section. The test modules also include a short series of introductory modules forming a mini-tutorial on how to use VV and some of the basics of Verilog.

VV’s interface contains the following components, 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 selecting 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 time slot. E.g., in the screenshot in Fig. 3, there are two blue-marked active events. Clicking one of the blue events 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, execution can be restarted by re-parsing the module.

**Implementation of VV.** We have implemented VV in ReScript [6], an OCaml dialect of JavaScript. We have used React [5] for the front-end of VV and Ohm [4] 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. The top-level state structure of VV, called state, containing, among other fields, the Verilog event queue, is defined as follows:9

```plaintext
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> }
```

9The actual event data type contains event_idx as well, this is only used for React-based visualisation and does not affect the semantics.
module interleave;
logic a, b;
always @ (a)
b = a;
initial begin
a = 0;
// ...
a = 1;
end
endmodule

module continterleave;
logic a;
wire b;
assign b = a;
initial begin
assign b = a;
end
initial begin
$display("b = %b", b);
a <= 1;
$display("b = %b", b);
end
endmodule

module binterleave;
logic a;
initial begin
a <= 0;
a <= 1;
end
endmodule

module nbinterleave;
logic a;
initial begin
a <= 0;
a <= 1;
end
endmodule

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, /*...*/)> }

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 full data type contains a few more fields not interesting enough to mention here and are therefore omitted in the presentation here (// [...]).

The core contribution of VV is its implementation of the Verilog event queue. 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 (that is, 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 monitors invocations will be 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 the reference algorithm): “update” events and “evaluation” events. In VV, the two categories are refined into six event types in total, as the event data type. The numbers of event types and the structure of each event type are not in any way canonical, instead, the two are largely a consequence of how other components of the interpreter are implemented; that is, other interpreters/semantics might end up with other refinements.

We now give a short overview of the event types in VV. 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, as discussed below (major problem two). The nba field of time_slot will only ever contain EventNBA events and, similarly, Events will only ever contain EventNBA events.

Major problem one: interleaving of processes. The first major problem we have run into when implementing VV is that the standard seems to allow for “too many” interleavings between processes. The problem arises from the fact that the standard states that “statements without time-control constructs in procedural blocks do not have to be executed as one event” [17, p. 66]. To exemplify, consider the module interleave in Fig. 4. The module illustrates a situation that can occur in hardware models implemented in Verilog: e.g., consider a module where the code of the two processes in the interleave module are used in two different processes modelling combinational logic (that is, stateless logic, which, according to the Verilog synthesis conventions should use blocking assignments). Now, if processes are allowed to interleave at any point, the following unfortunate interleaving would be allowed. First, the a = 0 assignment from the second process is executed, waking up the first process, which executes b = a and then preempts. Then, the first process executes a = 1 and terminates, the first process then continues
execution by going into waiting directly. Effectively, the first process misses the second update to the variable \(a\). This is not the behaviour we want if we are modelling, e.g., combinatorial logic (because for such logic, \(b\) should be equal to \(a\) after execution).

Hence, contrary to what the standard suggests, not all interleaving can be allowed. At the same time, if we look at existing simulators, it is not the case that no interleavings ever occur. More precisely, we are not aware of any situations in where two procedural processes are preempted and interleaved in existing simulators, but we are aware of situations where procedural processes are interleaved with continuous assignments. One such situation is illustrated by the `cont interleave` module in Fig. 4. Some existing simulators print \(b = x\) followed by \(b = x\) whereas other existing simulators print \(b = x\) followed by \(b = 1\). We are not aware of problems caused by interleaving procedural processes and continuous assignments.

One simple potential solution to the above is to disallow all interleavings, except for when a process blocks. Awaiting a clarified standard, this is the semantics VV implements. An alternative potential solution is to allow procedural processes and continuous assignments to interleave (either arbitrarily or only directly at the time of the update of the one of the continuous assignments’ dependencies), as currently done by some simulators.

**Major problem two: semantics of nonblocking assignments.** The second major problem we have run into is that the pseudocode function `execute_time_slot` in the reference algorithm is misleading. Specifically, in what semantics the pseudocode function suggests for nonblocking assignments. This is for two reasons, which we now give.

For illustration of the first reason, consider the two modules `binterleave` and `nbinterleave` in Fig. 4. If interleavings of procedural processes are disallowed, then `binterleave` will only ever print nothing or \(a = 1\), never \(a = 0\). Now, consider what output should be allowed for the `nbinterleave` module. In VV, executing the first block in `nbinterleave` schedules two events in the NBA region of the current time slot, specifically, the `nba` field of the current time slot will be `[EventNBA("a", 0), EventNBA("a", 1)]` after execution. After the second block has entered its waiting state, according to `execute_time_slot`, since both the active and inactive fields are empty, the contents of the `nba` field should be moved to the active fields. This is, however, not the full story, since the standard also says “NBAs shall be performed in the order the statements were executed” [17, p. 66]. Hence, it is not enough to simply move the events from the NBA region to the active region, since a defining characteristic of the active region is that its events are allowed to execute in any order. Therefore, additional structure is needed to keep track of NBA order dependencies in the queue. There are, of course, multiple ways to keep track of these order dependencies: for VV, we adapt the solution proposed by Meredith et al. [49] in their work on Verilog semantics, i.e., we keep track of the order dependencies by grouping all NBA events in the NBA region into a group event containing all the NBA events and move the group event to the active region instead of each individual NBA event. In VV, we call the group event `Events`, see the definition of the `event` data type above. Executing an `Events` event removes the first (sub)event from the event but leaves the event itself, unless the list of events was a singleton list, in which case the event is removed after execution. All in all, continuing the example, in VV, after moving the NBA events to the active region, the NBA region is empty and the active region is `Events(Events(Events(Events(Events(Events([... 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 (such as the execution of the `$display` statement in our example). 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.

In VV, whether the execution of NBA events can be interleaved by other types of events is controlled by the checkbox “Process NBA events first” in the left-most column of the interface. It is checked by default, not allowing NBA events to be interleaved.

To summarise, `execute_time_slot` suggests that executing NBA events is a simple matter of moving all NBA events from the NBA region to the active region, however, as we now have seen, executing NBA events is more involved.

---

11Another unrelated problem: Note that the quoted text has not been updated for SystemVerilog, specifically, assignments in the reactive region set execute after assignments 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 VV.
6. Additional Verilog commentary

We now discuss some, what we call, pragmatic shortcomings of Verilog. We discuss these shortcomings to clarify that not all oft-cited Verilog grievances cause problems for formalisation. These pragmatic shortcomings of Verilog differ from, what we here call, the semantic shortcomings of Verilog that we discussed in the previous section. By semantic shortcomings, we mean problems related to the meaning of Verilog arising from the Verilog standard being imprecise or otherwise unclear, e.g., as with the two problems we raised in the previous section. By pragmatic shortcomings, we mean commonly misunderstood aspects of Verilog that do not necessarily arise from a problem with the standard, but rather Verilog’s, at times, antipedagogical design. Pragmatic shortcomings can be resolved by a careful reading of the standard, whereas semantic shortcomings can not.

X values. Although difficult to use and make sense of intuitively,\(^ {12}\) 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. (Recall, 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.) 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).\(^ {13,14}\)

Interestingly, previous work on the formal semantics of Verilog seems to shy away from nets, in particular multidriven nets. Of particular interest here, Meredith et al. [49] work gets the distinction between variables and nets wrong, see Sec. 8.

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 `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. See the test modules under `init/` in VV for more discussion.

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, 16]). Whereas blocking assignments can be explained by stating that they have similar semantics as 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). VV shows how to think of assignments in terms of Verilog’s event queue. See, e.g., the test modules under `seq/` and `xx_more/Cummings/` in VV.

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

\(^{12}\) That X values are a common source of frustration is well-documented [50, 56, 58]. 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 utilised in the symbolic execution literature [28]. 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 [17, 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 [17, p. 299]).

\(^{13}\) The most upvoted question is a now-closed question about best coding practices for hardware-description languages (such as, of course, Verilog).

\(^{14}\) 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), not that variables with the data type will necessarily be mapped to storage elements 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.
continuous assignment updates [17, 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. We discuss some of them in the test modules cont/cont_delay1.sv, cont/cont_delay2.sv, etc. available in VV. 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. Related work: Verilog simulators

We now compare VV to existing Verilog simulators, which we here call traditional simulators, to differentiate them from VV, which is a visual simulator. Multiple traditional Verilog simulators exist today, such as the simulators shipped with large commercial hardware development IDEs such as Xilinx Vivado [63] and the open-source simulator Icarus Verilog [3] and Verilator [7].

VV and traditional simulators have different design goals; specifically, the design goal of VV is to explain the semantics of Verilog whereas the design goal of traditional simulators is to allow testing and debugging of (real-world) hardware designs. The two goals are similar but not identical. As a result, the two types of simulators end up differing substantially. We highlight key differences along three dimensions: simulator internals, human-simulator interaction, and analysis facilities.

Simulator internals. For traditional simulators, simulation speed is the main driving force behind the design of the simulator. In contrast, for VV, simulation speed is largely unimportant. Instead, the design of VV is driven the aim to provide accurate visualisations of all internals of executions of Verilog modules – internals traditionally not exposed by Verilog simulators (in particular, the event queue). The internals of VV must therefore be as conceptually/ontologically faithful to the standard as possible, even at the expense of efficiency.

Human-simulator interaction. Traditional simulators are batch tools, whereas VV is an interactive tool. To exemplify, recall that, due to concurrency, Verilog is a nondeterministic language. E.g., when there is a choice of multiple interleavings during execution, a traditional simulator will not explore all possible execution paths, but instead commit to one specific interleaving (which gives the best performance payoff) and ignore the remaining ones. In contrast, in VV it is up to the user to pick which particular interleaving to explore, enabling users to explore the full semantics of Verilog. (In this aspect, VV has more in common with explicit-state model checking than traditional simulators, except that VV is driven by human rather than machine.)

Analysis facilities. The analysis facilities offered by traditional simulators, in the form of debugging facilities, are designed to help its users to understand and find bugs in Verilog designs, whereas the analysis facilities offered by VV are designed for explaining the semantics of Verilog. Common debugging facilities in traditional simulators include “printf debugging” (e.g., $display and $monitor) and waveform visualisation – see Fig. 5 for an example waveform output from the simulator Icarus Verilog. Critically, in traditional simulators, the event queue cannot be inspected whereas in VV one of the main functions of the tool is to visualise the event queue.\[15\]

8. Related work: Verilog semantics

We now discuss some previous semantics work closely related to the work in this paper.

Verilog formalisations. Meredith et al.’s [49] Verilog semantics, implemented in the K framework [55], is to date the most complete Verilog semantics available. Their semantics follows the standard closely, specifically, the Verilog-2005 standard [8]. Like us, they omit support for multiple modules. Different from us, their semantics does not support X and Z values. More importantly, Meredith et al. have misunderstood the difference between variables and nets.\[16\] as a result do not include proper support for nets in their semantics. Thereby, they (unwittingly) do not take into consideration net-specific features such as multidriven nets. They, moreover, do not address the two problems we raise in Sec. 5; we illustrate the resulting problems exhibited by their semantics with test modules written for their semantics, see the k-verilog directory.

\[15\]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” [17, Ch. 36], do, as far as we are aware, allow for such inspection.

\[16\]See, e.g., the discussion on variables and nets in Sec. 4.1 of their paper, where they claim that “[t]he distinction between nets and variables is an archaic part of the standard, no longer strictly necessary”. Moreover, in Sec. 4.3, they claim that “[t]he best we can glean from the standard is that a net assignment should perform essentially as an always block with one blocking assignment in it”, which is clearly not the case since variables and nets have completely different evaluation models.
in the source code repository of VV. See also the modules under xx_more/Meredith/ in the drop-down menu with test modules in VV, which further discusses the questions for the Verilog community raised by Meredith et al. in their paper.

Meredith et al. emphasise that their semantics is executable. Executability here is a consequence of the semantics being implemented in the K framework, which allows for generation of, among other things, interpreters and model checkers. The K framework is a toolbox potentially perfect for the experienced semanticist, but potentially less perfect for those uninitiated in the field of programming-language semantics; or, put in comparative language, we believe our Verilog visualisation to be more approachable than K and a useful complement for Verilog users less experienced in formal semantics.

Bowen et al. [31] similarly emphasise executability (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 [38] early work on Verilog semantics 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, diverges ontologically from the standard): 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. and Gordon. We consider those projects to be either suggestions for alternative semantics for Verilog (rather than formalisations of Verilog) 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 [46] has developed DigitalJS, “a visual Verilog simulator for teaching”. DigitalJS uses Yosys [62] 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. Vivado. (On the topic on visualising synthesis algorithms, although not directly related to Verilog, Nestor [51] 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 [41] work on visualising stack-based concatenative languages (which, as Homer remarks, has sometimes (honorously) been referred to as “write-only” languages), Greenberg and Blatt’s [39] Shtepper which visualises the execution of shell scripts, Eisenstadt and Brayshaw’s [36] Prolog visualisation work, and the lambda calculus visualisations collected by Pramod’s [53]; visualisations of difficult corners of mainstream languages, e.g., the visualisation tool Loupe [54] which helps users “understand how JavaScript’s call stack/event loop/callback queue interact with each other” and Cooper’s [33] visualisations of coroutine event loops in JavaScript; and visualisations of distributed systems [29].

In the formal-methods literature, one can find examples of visualisations of formal-method techniques (e.g., so-called ribbon proofs for separation logic [61] or graphical proof assistants, designed for education, such as Holbert [52]) and examples of providing understandable presentations of proof states in program-verification tools (e.g., the proof-state visualisations in KeY’s [27] and Iris’ proof mode [43,44]).

**9. Conclusion**

With the ultimate aim of a mathematical formalisation of Verilog, we have introduced VV, a visualisation tool for Verilog’s simulation semantics. VV makes inspectable not only the parts of Verilog execution inspectable in traditional Verilog tools but also Verilog’s internal event queue. We understand VV as a visual formalisation of Verilog, and by visually formalising difficult-to-understand corners of Verilog in a form understandable by both hardware designers and programming-language-theory researchers, we hope our formalisation will be useful as a steppingstone in future work on a developing a mathematical formalisation of Verilog based on the Verilog standard.

**Acknowledgements** We thank Yann Herklotz and Oskar Abrahamsson for comments on drafts of this paper and/or testing the tool.

**References**
