To appear in *Proceedings of the Workshop on Formal Techniques for Hardware and Hardware-like Systems*, Marstrand, Sweden, June 1998.

Formal Reasoning with Verilog HDL

Gordon J. Pace[1] and Jifeng He[2]

[1] University of Malta, Msida MSD 06, Malta
[2] University of Oxford, Parks Road, Oxford, UK

Abstract. Most hardware verification techniques tend to fall under one of two broad, yet separate caps: simulation or formal verification. This paper briefly presents a framework in which formal verification plays a crucial role within the standard approach currently used by the hardware industry. As a basis for this, the formal semantics of Verilog HDL are defined, and properties about synchronization and mutual exclusion algorithms are proved.

1 Introduction

Hardware verification tends to be viewed by people from different backgrounds as being either exclusively simulation or formal verification. Combining the two together can provide a very powerful working environment, which provides both the flexibility and concept grasping of simulators and the rigorous background of formal methods [3].

Simulation allows faster development of design and cheaper and easier debugging during the design stage than if formal verification is used. The proliferation of standard hardware description languages (HDLs) made standard libraries of hardware components widely available and has made simulation approaches even more attractive. However, the loosely defined semantics of these HDLs makes formal verification (on their basis) impossible. Simulation on its own is not viable since it can only show the presence of errors, not their absence. Furthermore, the semantics of HDLs are usually much more involved than industry standard sequential software languages. Issues like parallel composition and non-determinism, play an important role in HDLs, and simulator runs do not do justice to these more complex concepts. For example, non-determinism in choosing which parallel thread to follow (from a number of enabled ones) is sometimes resolved in a simulator by taking the first one available. This may mean that no matter how many times a design is simulated, the result will invariably be the same, whereas the HDL semantics would allow an alternative sequence of execution which could end up with a different result.

Formal methods cannot however, replace existent methods of hardware design overnight. Figure 1 proposes one possible framework to gradually introduce formal methods in hardware design. The approach is completely built upon formal techniques but includes simulation for design visualization and development. Formal laws helping hardware engineers to correctly transform specifications into the implementation language are used to develop implementations guaranteed to be correct. These laws would not cover all possible case, but a simulator (guaranteed to have the same semantics as the interpretation given to the implementation language), can be used to develop implementations from the remaining specification portions, before they can be formally proved to be correct. The result is more reliability within an environment which does not require a complete revolution over the current trends.

This paper presents only one small part of this framework. The formal semantics of Verilog are defined as a formal basis for the whole strategy. The semantics are then used to prove properties of two small case studies involving synchronization and mutual exclusion issues.

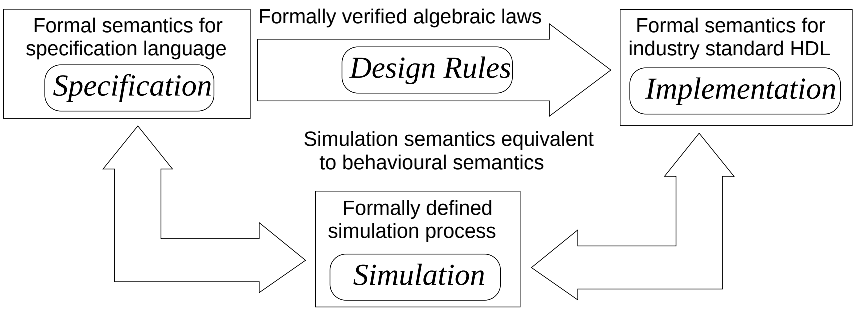

Fig. 1. Simulation and verification: a unified approach

2 VHDL and Verilog

Until now, most work done in the formalization of industry standard hardware simulation languages has almost exclusively dealt with VHDL[1, 2, 4, 5, 9, 11]. The main problem is that all of this research refers back to the original informal semantics description in the official documentation, leading to possible discrepancies. In comparison, until quite recently, much less work has been done on formal methods used with Verilog. The experience gained with VHDL should help this research to be more directed and to yield better results than would otherwise be expected. Hopefully, this will also reduce the diversity of methods used by different research groups and thus increase the rate at which information is built up. In [6], M.J.C. Gordon presented a semi-informal description of the semantics of Verilog preceding most major work on the language. This may help to provide a common stepping-stone out of the informal documentation which researchers may choose to make use of. Most importantly, however, one must realise that experience gained in either of the languages can usually be applied to the other. Thus, we believe that the main emphasis should be placed on the application of the formal semantics of these languages and techniques based on this kind of approach.

3 The Semantics of Verilog

3.1 Specification Organization

```
module NOT_beh (in, out);
input in; output out;
      forever  @in out = ~in;
endmodule
```

```
module NOT_struc (in, out);
input in; output out; wire p,g;
      POWER  P1  (p);
      GROUND G1  (g);
      PTRANS PT1 (p,in,out);
      NTRANS NT1 (g,in,out);
endmodule
```

Fig. 2. Behavioural and structural description of a negation gate

A Verilog specification is a closed system made up of a number of modules all of which run in parallel. Verilog allows both behavioural and structural descriptions of modules. Behavioural information, describing the behaviour of the outputs of a module, is what we will be mainly concerned with. Rather than define the behaviour, a structural description shows how the module is organised into sub-components. This approach is much more physically oriented. Figure 2 contrasts a behavioural with a structural description of a negation gate.

3.2 Approach Taken

The semantics of Verilog are described using a variant of Discrete Duration Calculus[14, 12, 8]. Rather than using the normal chop operator we define an alternative sequential composition operator which takes into consideration non-stable states. The resulting Relational Duration Calculus is similar to [13] and [10].

Duration Calculus. Only a brief overview of Duration Calculus (DC) will be given here. Readers interested in fuller accounts of the calculus may refer to [7].

Boolean States and Expressions: DC is a temporal logic allowing one to state properties pertaining to time in a straightforward and natural fashion. The basic building blocks of the calculus are *state variables*, functions from time to the boolean values 1 and 0. Time is assumed to be represented by the non-negative real numbers. **1** and **0** are defined to be the constant state functions: always true and always false respectively.
State variables can be combined together using standard boolean connectives to form *state expressions*. The interpretation of such expressions is simply to take the boolean operator pointwise on the operands. Thus, for example, at any time t:

$$(P \wedge Q)(t) \stackrel{def}{=} P(t) \wedge Q(t)$$

Duration Formulae: However, as the name of the calculus suggests, DC deals with interval (or state duration) reasoning, rather than remaining at the pointwise reasoning level. Reasoning is thus promoted to functions from time intervals (of the form $[s, f]$) to the boolean values. Such functions are called *duration formulae.*

If P is a state expression:

$$(\int P = n)[s, f] \stackrel{def}{=} \int_s^f P(t)\mathrm{d}t = n$$

$D; E$, where ; is the chop (or sequential composition) operator, states that the time interval can be split into two consecutive ones such that D holds over the first interval and E over the second:

$$(D; E)[s, f] \stackrel{def}{=} \exists m \in [s, f] \cdot D[s, m] \wedge E[m, f]$$

Again, the boolean operators are overloaded to act over duration formulae using a similar definition as before. Hence,for any time interval $[s, f]$:

$$(D \wedge E)[s, f] \stackrel{def}{=} D[s, f] \wedge E[s, f]$$

From these basic operators, a variety of other useful operators can be defined:

- For a state expression P, $\lceil P \rceil$ holds whenever P is true over the whole, non-empty interval:
$$\lceil P \rceil[s, f] \stackrel{def}{=} \int P = f - s \ \wedge \ f > s$$
- $\lceil\rceil$ is true only for empty intervals:
$$\lceil\rceil \stackrel{def}{=} \neg\lceil \mathbf{1} \rceil$$
- **true** and **false**, the two constant duration formulae, can now be defined as $(\lceil\rceil \vee \lceil \mathbf{1} \rceil)$ and $(\neg\mathbf{false})$ respectively.
- The length of the interval l can be defined as:
$$l = n \stackrel{def}{=} \int \mathbf{1} = n$$
Note that $l \geq n$ can be defined as $l = n; \mathbf{true}$. From these, $l < n$, $l > n$ and $l \leq n$ can then be defined.
- The standard temporal logic operators $\Box$ and $\Diamond$ can also be easily defined. $\Diamond D$, read as 'sometimes D' is true over an interval $[s, f]$, provided that duration formula D holds over some sub-interval of the one in question. $\Box D$, read as 'always D' is true provided that D holds over all sub-intervals.

$$\Diamond D \stackrel{def}{=} \mathbf{true}; D; \mathbf{true}$$
$$\Box D \stackrel{def}{=} \neg\Diamond\neg D$$

$\lceil P \rceil^*$ has the same meaning as $\lceil P \rceil$ but also allows for the possibility of an empty interval.

$$\lceil P \rceil^* \stackrel{def}{=} \lceil\rceil \vee \lceil P \rceil$$

Discrete Duration Calculus. Discrete Duration Calculus is simply a restricted version of the continuous one. The restrictions are:

- Any discontinuities in the boolean states belong to $\mathbb{N}$

- Duration formulae act on intervals $[b, e]$, where both b and e are in $\mathbb{N}$

Note that in Discrete DC, the shortest non-zero interval is of length 1. This encourages the definition of another operator:

$$\lceil\lceil P \rceil\rceil \stackrel{def}{=} \lceil P \rceil \wedge l = 1$$

Relational Duration Calculus. Three new operators are described below:
Pre-value of a state variable: $\overleftarrow{P}$ is a temporal formula giving the value of P just before the start of the interval.

$$(\overleftarrow{v} = x)[b, e] \stackrel{def}{=} \begin{cases} \text{false} & \text{if } b = 0 \\ \lim_{t \to b^-} v(t) \text{ is equal to } x & \text{otherwise} \end{cases}$$

Post-value of a state variable: $\overrightarrow{v}$ is the 'dual' of $\overleftarrow{v}$. It is defined as the right limit of v at the end point of the interval:

$$(\overrightarrow{v} = x)[b, e] \stackrel{def}{=} \lim_{t \to e^+} v(t) \text{ is equal to } x$$

Relational Chop: $D \overset{W}{;} E$ is the sequential chop operator which behaves just like the normal sequential composition, but with the extra restriction that the pre-values of state variables W in E are the same as their post-values in D.

$$\begin{aligned} D \overset{W}{;} E \stackrel{def}{=} & \\ & \exists W' : \mathbb{B}^* \cdot \exists W_D, W_E \cdot \\ & (D[W_D/W] \wedge \bigwedge_{v \in W} (v' = \overrightarrow{v_D} \wedge \overleftarrow{v_D} = \overleftarrow{v} \wedge \lceil v \Leftrightarrow v_D \rceil^*)); \\ & (E[W_E/W] \wedge \bigwedge_{v \in W} (\overleftarrow{v_E} = v' \wedge \lceil v \Leftrightarrow v_E \rceil^* \wedge \overrightarrow{v} = \overrightarrow{v_E}) \end{aligned}$$

Most of the algebraic laws for the normal chop operator still hold for this relational chop. Despite the apparent complexity of the definition, we found that manipulation of expressions using relational chop is not more complex than using the normal chop operator, mainly thanks to the algebraic laws which we have derived. This is, however, beyond the scope of this paper, and will not be investigated further here.

3.3 Modules

We assume that each module P has a number of output wires $\mathcal{O}_{\mathtt{P}}$ to which no other module may write. Also, the assignments to the outputs of a module must take some time. All modules are allowed to read the output variables of other modules, but reading and writing to and from the same global wires at the same time is not permitted to avoid non-determinism.

The assumption that module outputs are disjoint gives us the opportunity to define parallel composition as:

$$[\![\texttt{P} \parallel \texttt{Q}]\!] \stackrel{def}{=} [\![\texttt{P}]\!] \wedge [\![\texttt{Q}]\!]$$

Continuous assignment: `assign v=e` forces `v` to the value of expression `e`:

$$[\![\texttt{assign v=e}]\!] \stackrel{def}{=} \lceil v = e \rceil^*$$

where $\lceil P \rceil^* = \lceil P \rceil \vee \lceil\rceil$ (either P is true over the interval, or it is an empty interval).

Procedural behaviour: `initial P` behaves like the sequential program `P`:

$$[\![\texttt{initial P}]\!] \stackrel{def}{=} [\![\texttt{P}]\!]_{\mathcal{O}_\texttt{P}}(\mathbf{Const}(\mathcal{O}_\texttt{P}))$$

$[\![\texttt{P}]\!]_W(D)$ describes the behaviour of an individual program module `P` whose output wires are given in set W and which will, upon termination, behave as described by the duration formula D. $\mathbf{Const}(W)$ is defined as follows:

$$\mathbf{Const}(W) \stackrel{def}{=} \forall w \in W \cdot \exists b \cdot (\overleftarrow{w} = b) \wedge (\overrightarrow{w} = b) \wedge \lceil w = b \rceil^*$$

3.4 Imperative Programming Statements

Verilog statements can be split into two sets: imperative programming-like constructs which take no simulation time, and timing control instructions, which are closer to hardware concepts and may take simulation time to execute.

Assignments:	$[\![\texttt{v=e}]\!]_W(D) \stackrel{def}{=} (\overrightarrow{v} = \overleftarrow{e} \wedge \mathbf{Const}(W - \{v\}) \wedge \lceil\rceil) \overset{W}{\fatsemi} D$
Conditionals:	$[\![\texttt{if b then P else Q}]\!]_W(D) \stackrel{def}{=} [\![\texttt{P}]\!]_W(D) \triangleleft \overleftarrow{b} \triangleright [\![\texttt{Q}]\!]_W(D)$
Sequential composition:	$[\![\texttt{P;Q}]\!]_W(D) \stackrel{def}{=} [\![\texttt{P}]\!]_W([\![\texttt{Q}]\!]_W(D))$
Loops:	$[\![\texttt{while b do P}]\!]_W(D) \stackrel{def}{=} \mu X \cdot ([\![\texttt{P}]\!]_W(X) \triangleleft \overleftarrow{b} \triangleright D)$

To avoid problems with programs such as `while true do skip`, we insist that bodies of loops must take time to terminate. A simple syntactic check is usually sufficient to ensure this.

The semantics of forever loops, case statements, etc can be specified in terms of these constructs.

3.5 Timing Control Instructions

Blocking Assignments. Assignments can be delayed by using guards, which block time until a certain condition is satisfied. The assignment `v=guard e` reads the value of expression `e` and assigns it to `v` as soon as the guard is lowered:

$$[\![\texttt{v=guard e}]\!]_W(D) \stackrel{def}{=} \exists \alpha \cdot [\![\alpha\texttt{=e ; guard ; v=}\alpha]\!]_{W \cup \{\alpha\}}(D)$$

The assignment `guard v=e` waits until the guard is lowered, reads the value of expression `e` and assigns it to `v`:

$$[\![\texttt{guard v=e}]\!]_W(D) \stackrel{def}{=} [\![\texttt{guard ; v=e}]\!]_W(D)$$

Guards. Guards control the flow of time by blocking further execution until they are lowered. Two types of guards are treated here: time delay guards and level triggered guards. Other types of guards can be described in a similar manner.

`#n` blocks the execution of a module by `n` time units:

$$\begin{aligned} [\![\texttt{\#n}]\!]_W(D) \stackrel{def}{=} \; & (l < n \land \mathbf{Const}(W)) \lor \\ & (l = n \land \mathbf{Const}(W)) \overset{W}{;} D \end{aligned}$$

`wait v` blocks execution until v carries the value 1.

$$\begin{aligned} [\![\texttt{wait v}]\!]_W(D) \stackrel{def}{=} \; & (\lceil \neg v \rceil^* \land \neg \overrightarrow{v} \land \mathbf{Const}(W)) \lor \\ & (\lceil \neg v \rceil^* \land \overrightarrow{v} \land \mathbf{Const}(W)) \overset{W}{;} D \end{aligned}$$

Spikes on communication variables are considered to be undesirable behaviour and are not captured by `wait` statements. A syntactic check usually suffices to ensure that no spikes will appear on a global variable in the system.

4 Two Case Studies

To demonstrate the use of these semantics, we take two standard examples of programs using variables for synchronization or mutual exclusion.

4.1 Program Control Variables

To prove properties pertaining to synchronization and mutual exclusion, it is necessary to be able to discuss where control in a program resides. This can be done by using auxiliary variables. If the program in question is P, we define two auxiliary variables s_P and f_P representing 'started P' and 'finished P'. Both variables are initialized to 0 and the program portion P is enclosed within two assignments:

$$s_P = 1; P; f_P = 1$$

Thus, for example, P is active over a time interval if $\lceil s_P \land \neg f_P \rceil$. We assume that P does not occur inside a loop.

An alternative is to use only one variable which specifies whether P is active or not. However, synchronization properties can be more easily expressed in the notation we choose. Furthermore, in the alternative notation, most properties become impossible to specify for point programs (which do not take physical time to execute), since they appear to be permanently inactive.

The following laws about start and termination control variables will be found useful in later proofs:

Once started, always started:	$\Box(\lceil s_P \rceil; \mathbf{true} \Rightarrow \lceil s_P \rceil)$
Not started, never started:	$\Box(\mathbf{true}; \lceil \neg s_P \rceil \Rightarrow \lceil \neg s_P \rceil)$
Once finished, always finished:	$\Box(\lceil f_P \rceil; \mathbf{true} \Rightarrow \lceil f_P \rceil)$
Not finished, never finished:	$\Box(\mathbf{true}; \lceil \neg f_P \rceil \Rightarrow \lceil \neg f_P \rceil)$
Start before termination:	$\lceil \neg(\neg s_P \land f_P) \rceil^*$

4.2 Specification Language

When certain specification properties occur frequently, it is more effective to define a specification language to express the properties more concisely. Algebraic laws of the specification language can then be effective engineering tools in proofs.

Non-overlapping Processes. Given two program portions, P and Q, they are said to be non-overlapping if they do not share any common execution time. This will be written as P $\mathsf{non\text{-}overlap}$ Q and is defined as:

$$P \ \mathsf{non\text{-}overlap}\ Q \stackrel{def}{=} \lceil \neg s_P \vee f_P \vee \neg s_Q \vee f_Q \rceil^*$$

Disjoint Processes. Two overlapping processes may have a common execution point if at the moment when the first finishes, the other starts. This may not always be desirable, for instance in the case when the second process reads a register that the previous process has just written to. A short delay between the execution of the processes ensures that the values written by the first program have stabilized and thus there is no concurrent read and write.

Two programs P and Q are said to be disjoint, written as P $\mathsf{disjoint}$ Q, if there is a delay between the termination of one and the commencement of the other:

$$\begin{array}{rcl} P \ \mathsf{disjoint}\ Q & \stackrel{def}{=} & P \ \mathsf{non\text{-}overlap}\ Q \quad \wedge \\ & & \begin{pmatrix} \Diamond\lceil f_P \rceil \\ \vee\Diamond\lceil f_Q \rceil \end{pmatrix} \Rightarrow \begin{pmatrix} \Diamond\lceil f_P \wedge \neg s_Q \rceil \\ \vee\Diamond\lceil f_Q \wedge \neg s_P \rceil \end{pmatrix} \end{array}$$

Synchronization. Three types of synchronization are considered:

- Synchronized start: P $\mathsf{synchro}_L$ Q is true if P and Q start at the same time.

$$P \ \mathsf{synchro}_L\ Q \stackrel{def}{=} \lceil s_P = s_Q \rceil^*$$

- Synchronized finish: P $\mathsf{synchro}_R$ Q is true if P and Q terminate at the same time.

$$P \ \mathsf{synchro}_R\ Q \stackrel{def}{=} \lceil f_P = f_Q \rceil^*$$

- Fully synchronized: P $\mathsf{synchro}$ Q is true if P and Q start and terminate together.

$$P \ \mathsf{synchro}\ Q \stackrel{def}{=} P \ \mathsf{synchro}_L\ Q \quad \wedge \quad P \ \mathsf{synchro}_R\ Q$$

Two laws about synchronization will be used. The first states that all initial programs start concurrently, and the second relates synchronization with sequential composition.

- If $[\![$`initial P`$]\!]$ and $[\![$`initial Q`$]\!]$, then:
$$P \ \mathsf{synchro}_L \ Q$$
- If $[\![$`initial P ; Q`$]\!]$ and $[\![$`initial R ; S`$]\!]$, then:
$$P \ \mathsf{synchro}_R \ R = Q \ \mathsf{synchro}_L \ S$$

4.3 Case Study: Synchronizing Handshake

The first example involves two processes using two variables to synchronize two portions of program running in parallel.

Each process has a variable which it sets to true once it is ready to execute the synchronized part. It then waits for the other program's variable to become true. Note that these variables are always initialized to zero.

$$\texttt{HS(x,y)} \stackrel{def}{=} \texttt{x=\#1 1; wait y; x=\#1 0;}$$

Now consider two parallel programs which use this synchronizing portion:

$$\begin{aligned}\texttt{COMM} \stackrel{def}{=} & \quad \texttt{(initial x=0; P1; HS(x,y); P2)} \\ & \| \, \texttt{(initial y=0; Q1; HS(y,x); Q2)}\end{aligned}$$

where variables `x` and `y` are not free in programs `P1`, `P2`, `Q1` and `Q2`.

Mutual Exclusion. The synchronization mechanism ensures that `P1` and `Q2` do not interfere (similarly `Q1` and `P2`) and do not share any common execution time. We first prove that `P1` non-overlap `Q2` and then use this result to show that `P1` disjoint `Q2`. The similar result for `Q1` and `P2` follows immediately by symmetry.

Lemma 1. `x` *starts off false and remains so for some time even after* `P1` *has terminated:*

$$\begin{gathered}\lceil \neg f_{P1} \wedge \neg x \rceil^* \vee \\ \lceil \neg f_{P1} \wedge \neg x \rceil^* ; \lceil f_{P1} \wedge \neg x \rceil ; \lceil f_{P1} \rceil^*\end{gathered}$$

Proof. The proof follows from the definition of the semantics and DC reasoning:

$\llbracket\texttt{COMM}\rrbracket$
$\Rightarrow$ { semantics of parallel composition }
$\llbracket\texttt{initial x,}s_{P1},s_{P2},f_{P1},f_{P2}\texttt{=0,0,0,0,0;}$
$\quad s_{P1}\texttt{=1; P1; }f_{P1}\texttt{=1; HS(x,y); }s_{P2}\texttt{=1; P2; }f_{P2}\texttt{=1}\rrbracket$
$\Rightarrow (\overrightarrow{x}=0 \wedge \overrightarrow{f}_{P1}=0\wedge\lceil\rceil) \overset{W}{\fatsemi}$
$\quad \llbracket s_{P1}\texttt{=1; P1; }f_{P1}\texttt{=1; HS(x,y); }s_{P2}\texttt{=1; P2; }f_{P2}\texttt{=1}\rrbracket_W(\mathbf{Const}(W))$
$\Rightarrow$ { $\texttt{P1}$ does not write to $\texttt{x}$ or f_{P1} }
$(\overrightarrow{x}=0 \wedge \overrightarrow{f}_{P1}=0\wedge\lceil\rceil) \overset{W}{\fatsemi} \mathbf{Const}(\{x,f_{P1}\}) \overset{W}{\fatsemi} (\overrightarrow{x}=\overleftarrow{x}\wedge\overrightarrow{f}_{P1}=1\wedge\lceil\rceil) \overset{W}{\fatsemi}$
$\quad \llbracket\texttt{HS(x,y); }s_{P2}\texttt{=1; P2; }f_{P2}\texttt{=1}\rrbracket_W(\mathbf{Const}(W))$
$\Rightarrow (\lceil\neg x\wedge\neg f_{P1}\rceil^*\wedge\overrightarrow{x}=0\wedge\overrightarrow{f}_{P1}=1) \overset{W}{\fatsemi}$
$\quad \llbracket\texttt{\#1 x=1; wait y x=\#1 0; }s_{P2}\texttt{=1; P2; }f_{P2}\texttt{=1}\rrbracket_W(\mathbf{Const}(W))$
$\Rightarrow (\lceil\neg x\wedge\neg f_{P1}\rceil^*\wedge\overrightarrow{x}=0\wedge\overrightarrow{f}_{P1}=1) \overset{W}{\fatsemi}$
$\quad (l<1\vee((l=1\wedge\mathbf{Const}(\{x,f_{P1}\}) \overset{W}{\fatsemi} \mathbf{true}))$
$\Rightarrow$ { $l<1$ equivalent to $\lceil\rceil$ in Discrete DC }
$\lceil\neg x\wedge\neg f_{P1}\rceil^*\vee$
$\lceil\neg x\wedge\neg f_{P1}\rceil^*;\lceil\neg x\wedge f_{P1}\rceil;\mathbf{true}$
$\Rightarrow$ { once finished, always finished }
$\lceil\neg x\wedge\neg f_{P1}\rceil^*\vee$
$\lceil\neg x\wedge\neg f_{P1}\rceil^*;\lceil\neg x\wedge f_{P1}\rceil;\lceil f_{P1}\rceil^*$

□

Lemma 2. *If* Q2 *has started,* x *must have been true for some time.*

$$\lceil\neg s_{Q2}\rceil^*\vee$$
$$\lceil\neg s_{Q2}\rceil;\lceil x\rceil;\mathbf{true}$$

Proof. The proof is similar to the previous one:

$\llbracket\texttt{COMM}\rrbracket$
$\Rightarrow$ { semantics of parallel composition }
$\llbracket\texttt{initial y,}s_{Q1},s_{Q2},f_{Q1},f_{Q2}\texttt{=0,0,0,0,0;}$
$\quad s_{Q1}\texttt{=1; Q1; }f_{Q1}\texttt{=1; HS(y,x); }s_{Q2}\texttt{=1; Q2; }f_{Q2}\texttt{=1}\rrbracket$
$\Rightarrow (\overrightarrow{s}_{Q2}=0\wedge\lceil\rceil) \overset{V}{\fatsemi} \mathbf{Const}(\{s_{Q2}\}) \overset{V}{\fatsemi}$
$\quad \llbracket\texttt{\#1 y=1; wait x; }\ldots\rrbracket_V(\mathbf{Const}(V))$
$\Rightarrow (\overrightarrow{s}_{Q2}=0\wedge\lceil\rceil) \overset{V}{\fatsemi} \mathbf{Const}(\{s_{Q2}\}) \overset{V}{\fatsemi}$
$(l<1\vee((l=1\wedge\mathbf{Const}(\{s_{Q2}\})) \overset{V}{\fatsemi} \llbracket\texttt{y=1; wait x; }\ldots\rrbracket_V(\mathbf{Const}(V))))$
$\Rightarrow \lceil\neg s_{Q2}\rceil^*\vee$
$(\lceil\neg s_{Q2}\rceil\wedge\overrightarrow{s}_{Q2}=0) \overset{V}{\fatsemi} \llbracket\texttt{wait x; }\ldots\rrbracket_V(\mathbf{Const}(V))$
$\Rightarrow \lceil\neg s_{Q2}\rceil^*\vee$
$\lceil\neg s_{Q2}\rceil;\lceil x\rceil;\mathbf{true}$

□

As can be surmised from the two lemmas just proved, these basic results can be quite easily, albeit tediously, proved. Proofs of the remaining lemmata in the paper will be omitted since they follow the same routine used in the proofs of lemmata 1 and 2.

Theorem 1. P1 non-overlap Q2

Proof. The proof follows from Lemmata 1 and 2:

$$
\begin{array}{l}
\quad \text{Lemma 1, Lemma 2 and } (A \vee B) \wedge (C \vee D) \Rightarrow A \vee C \vee (B \wedge D) \\
\Rightarrow \lceil \neg s_{Q2} \rceil^* \vee \lceil \neg f_{P1} \rceil^* \vee \\
\quad (\lceil \neg s_{Q2} \rceil ; \lceil x \rceil ; \mathbf{true} \;\; \wedge \;\; (\lceil \neg f_{P1} \wedge \neg x \rceil^* ; \lceil f_{P1} \wedge \neg x \rceil ; \mathbf{true}) \\
\Rightarrow \{ \text{ DC reasoning } \} \\
\quad \lceil \neg s_{Q2} \rceil^* \vee \lceil \neg f_{P1} \rceil^* \vee \\
\quad \lceil \neg s_{Q2} \rceil ; \lceil f_{P1} \rceil \\
\Rightarrow \lceil \neg s_{Q2} \vee f_{P1} \rceil^* \\
\Rightarrow \lceil \neg s_{P1} \vee f_{P1} \vee \neg s_{Q2} \vee f_{Q2} \rceil^* \\
= P \text{ non-overlap } Q
\end{array}
$$

□

Theorem 2. P1 disjoint Q2

Proof. The proof is split into two parts:

Part I:

$$
\begin{array}{l}
\quad \Diamond \lceil f_{P1} \rceil \\
= \mathbf{true} ; \lceil f_{P1} \rceil ; \mathbf{true} \\
\Rightarrow \{ \text{ lemma 1 } \} \\
\quad ((\lceil \neg f_{P1} \rceil^* ; \lceil f_{P1} \rceil) \wedge \lceil \neg x \rceil) ; \mathbf{true})) \\
\Rightarrow \{ \text{ lemma 2 } \} \\
\quad ((\lceil \neg f_{P1} \rceil^* ; \lceil f_{P1} \rceil) \wedge \lceil \neg s_{Q2} \rceil) ; \mathbf{true})) \\
\Rightarrow \mathbf{true} ; \lceil f_{P1} \wedge \neg s_{Q2} \rceil ; \mathbf{true} \\
= \Diamond \lceil f_{P1} \wedge \neg s_{Q2} \rceil
\end{array}
$$

Part II:

$$
\begin{array}{l}
\quad \Diamond \lceil f_{Q2} \rceil \\
\Rightarrow \mathbf{true} ; \lceil f_{Q2} \rceil ; \mathbf{true} \\
\Rightarrow \{ \text{ start before finish } \} \\
\quad \mathbf{true} ; \lceil s_{Q2} \rceil ; \mathbf{true} \\
\Rightarrow \{ \text{ lemma 2 and DC reasoning } \} \\
\quad \mathbf{true} ; \lceil x \rceil ; \mathbf{true} \\
\Rightarrow \{ \text{ lemma 1 and DC reasoning } \} \\
\quad \mathbf{true} ; \lceil f_{P1} \rceil ; \mathbf{true} \\
= \Diamond \lceil f_{P1} \rceil
\end{array}
$$

From Theorem 1 and parts I and II we can conclude that P disjoint Q.

□

Synchronization. Still considering the same parallel program, we would like to show some synchronization properties of its components. There are two interesting results we can show: that `P1;HS(x,y)` is fully synchronized with `Q1;HS(y,x)` and that `P2` and `Q2` have a synchronized start.

Lemma 3. *$(x \wedge y)$ must be true for exactly one time unit just before* `P1;HS(x,y)` *terminates.*

$$\lceil \neg f_{P1;HS(x,y)} \wedge \neg(x \wedge y)\rceil^* \vee$$
$$(\lceil \neg f_{P1;HS(x,y)}\rceil \wedge (\lceil \neg(x \wedge y)\rceil ; \lceil\lceil x \wedge y\rceil\rceil)); \lceil f_{P1;HS(x,y)}\rceil^*$$

Lemma 4. `(P1;HS(x,y))` $\mathsf{synchro}_L$ `(Q1;HS(y,x))`

Proof. Immediately true from the law about initial program segments. □

Lemma 5. `(P1;HS(x,y))` $\mathsf{synchro}_R$ `(Q1;HS(y,x))`

Proof. The proof uses Lemma 3:

$$
\begin{array}{ll}
 & \text{Lemma 3 applied to both processes} \\
\Rightarrow & \lceil \neg(x \wedge y) \wedge \neg f_{P1;HS(x,y)} \wedge \neg f_{Q1;HS(y,x)}\rceil^* \vee \\
 & \quad (\lceil \neg f_{P1;HS(x,y)}\rceil \wedge (\lceil \neg(x \wedge y)\rceil ; \lceil\lceil x \wedge y\rceil\rceil)); \lceil f_{P1;HS(x,y)}\rceil^* \wedge \\
 & \quad (\lceil \neg f_{Q1;HS(y,x)}\rceil \wedge (\lceil \neg(x \wedge y)\rceil ; \lceil\lceil x \wedge y\rceil\rceil)); \lceil f_{Q1;HS(y,y)}\rceil^* \\
\Rightarrow & \lceil \neg f_{P1;HS(x,y)} \wedge \neg f_{Q1;HS(y,x)}\rceil^* \vee \\
 & \lceil \neg f_{P1;HS(x,y)} \wedge \neg f_{Q1;HS(y,x)}\rceil ; \lceil f_{P1;HS(x,y)} \wedge f_{Q1;HS(y,x)}\rceil \\
\Rightarrow & \lceil f_{P1;HS(x,y)} = f_{Q1;HS(y,x)}\rceil^* \\
= & \texttt{(P1;HS(x,y))}\ \mathsf{synchro}_R\ \texttt{(Q1;HS(y,x))}
\end{array}
$$

□

Theorem 3. `(P1;HS(x,y))` $\mathsf{synchro}$ `(Q1;HS(y,x))`

Proof. Follows immediately from Lemmata 4 and 5. □

Theorem 4. `(P2)` $\mathsf{synchro}_L$ `(Q2)`

Proof. Follows from Lemma 5 and law about synchronization and sequential composition. □

4.4 Case Study: Mutual Exclusion

The second example involves two processes competing for a valuable resource which can only be used by one process at a time.

Each client process has a variable which it sets to true once it needs to use the resource. It then waits until another variable denoting the availability of the resource to be used by the process itself becomes true. A queue handler process loops forever giving control to the first process to ask for the resource. In the case of a tie, the first process is given priority.

```
QH = u1, u2 = 0, 0;
     forever begin
       wait (g1 or g2);
       #1 u1, u2 = g1, g2 and not g1;
       if (u1) then wait (not g1);
               else wait (not g2);
       #1 u1, u2 = 0, 0;
     end;
```

```
CL1 = g1 = 0;
      P1;
      g1 = #1 1;
      wait (u1);
      Q1;
      g1 = #1 0;
```

CL2 is defined similarly to CL1 but using variables g2 and u2 rather than g1 and u1 respectively. Processes P1, Q1, P2 and Q2 do not use the control variables.

Lemma 6. u1 *must be true when* Q1 *terminates.*

$$\begin{array}{c} \lceil \neg s_{Q1} \rceil^* \vee \\ \lceil \neg s_{Q1} \rceil^*; \lceil \neg f_{Q1} \wedge u1 \rceil \vee \\ \lceil \neg s_{Q1} \rceil^*; (\lceil u1 \rceil \wedge \mathbf{true}; \lceil f_{Q1} \rceil); \lceil f_{Q1} \rceil^* \end{array}$$

The symmetrical result for Q2 *will be referred to as Lemma 6+.*

Lemma 7. $\lceil \neg (u1 \wedge u2) \rceil^*$

Theorem 5. Q1 non-overlap Q2

Proof. The proof follows from Lemma 6 and its symmetric result.

$$\begin{array}{ll} & \text{Lemma 6 and lemma 6+} \\ \Rightarrow & \lceil (s_{Q1} \wedge \neg f_{Q1}) \Rightarrow u1 \rceil^* \wedge \\ & \lceil (s_{Q2} \wedge \neg f_{Q2}) \Rightarrow u2 \rceil^* \\ \Rightarrow & \{ \text{ lemma 7 } \} \\ & \lceil \neg s_{Q1} \vee f_{Q1} \vee \neg s_{Q2} \vee f_{Q2} \rceil^* \\ = & \texttt{Q1 non-overlap Q2} \end{array}$$

□

Theorem 6. Q1 disjoint Q2

Proof. The following result will be used later in the proof:

$$\begin{array}{ll} & \Diamond \lceil f_{Q1} \rceil \\ \Rightarrow & \{ \text{ lemma 6 and law about termination variables } \} \\ & \lceil \neg s_{Q1} \rceil^*; ((\mathbf{true}; \lceil f_{Q1} \rceil) \wedge \lceil u1 \rceil); \mathbf{true} \end{array}$$

Also from lemma 6+:

$$\begin{array}{c} \lceil \neg s_{Q2} \rceil^* \vee \\ \lceil \neg s_{Q2} \rceil^*; \lceil \neg f_{Q2} \wedge u2 \rceil \vee \\ \lceil \neg s_{Q2} \rceil^*; (\lceil u2 \rceil \wedge \mathbf{true}; \lceil f_{Q2} \rceil); \lceil f_{Q2} \rceil^* \end{array}$$

Considering the three possible duration formulae:

Case i:

$$\begin{array}{ll} & \lceil \neg s_{Q2} \rceil^* \\ \Rightarrow & \{ \text{ from previous result } \} \\ & \mathbf{true}; \lceil f_{Q1} \wedge \neg s_{Q2} \rceil; \mathbf{true} \\ = & \Diamond \lceil f_{Q1} \wedge \neg s_{Q2} \rceil \end{array}$$

Case ii:

$$\begin{array}{ll} & \lceil \neg s_{Q2} \rceil^*; \lceil \neg f_{Q2} \wedge u2 \rceil \\ \Rightarrow & \lceil \neg s_{Q2} \rceil^*; \lceil u2 \rceil \\ \Rightarrow & \{ \text{ lemma 3.2 } \} \\ & \lceil \neg s_{Q2} \rceil^*; \lceil \neg u1 \rceil \\ \Rightarrow & \{ \text{ from previous result } \} \\ & \mathbf{true}; \lceil f_{Q1} \wedge \neg s_{Q2} \rceil; \mathbf{true} \\ = & \Diamond \lceil f_{Q1} \wedge \neg s_{Q2} \rceil \end{array}$$

Case iii:

$$
\begin{array}{ll}
& \lceil\neg s_{Q2}\rceil^*; (\lceil u2\rceil \wedge \mathbf{true}; \lceil f_{Q2}\rceil); \lceil f_{Q2}\rceil^* \\
\Rightarrow & \{\text{ previous reasoning and lemma 3.2 }\} \\
& \lceil\neg s_{Q2}\rceil^*; (\lceil u2\rceil \wedge \mathbf{true}; \lceil f_{Q2}\rceil); \lceil f_{Q2}\rceil^* \wedge \\
& \lceil\neg s_{Q1}\rceil^*; ((\mathbf{true}; \lceil f_{Q1}\rceil) \wedge \lceil\neg u2\rceil); \mathbf{true} \\
\Rightarrow & \{\text{ DC reasoning}\} \\
& (\lceil\neg s_{Q2}\rceil^* \wedge (\mathbf{true}; \lceil f_{Q1}\rceil; \mathbf{true})); \mathbf{true} \vee \\
& (\lceil\neg s_{Q1}\rceil^* \wedge (\mathbf{true}; \lceil f_{Q2}\rceil; \mathbf{true})); \mathbf{true} \\
\Rightarrow & \mathbf{true}; \lceil f_{Q1} \wedge \neg s_{Q2}\rceil; \mathbf{true} \vee \\
& \mathbf{true}; \lceil f_{Q2} \wedge \neg s_{Q1}\rceil; \mathbf{true} \\
= & \Diamond\lceil f_{Q1} \wedge \neg s_{Q2}\rceil \vee \\
& \Diamond\lceil f_{Q2} \wedge \neg s_{Q1}\rceil
\end{array}
$$

Symmetrical reasoning holds for the case when $\Diamond\lceil f_{Q2}\rceil$. Hence, it has been proved that:

$$
\begin{pmatrix} \Diamond\lceil f_{Q1}\rceil \\ \vee\Diamond\lceil f_{Q2}\rceil \end{pmatrix} \Rightarrow \begin{pmatrix} \Diamond\lceil f_{Q1} \wedge \neg s_{Q2}\rceil \\ \vee\Diamond\lceil f_{Q2} \wedge \neg s_{Q1}\rceil \end{pmatrix}
$$

This result, together with Theorem 3.1 shows that `Q1 disjoint Q2`.

□

5 Conclusions and Future Work

Both case studies presented here would benefit from further generalization. Using induction, both algorithms can be readily extended to work with more than two processes. In the first case study, generalizing to processes which use more than a single occurrence of the handshake code is also desirable. Finally, proving fairness in the second case study is also an interesting challenge. It is desirable to show that as long as all code inside crucial sections terminates, asking for the resource guarantees that it is eventually made available. Intuitively, the delays used in the assignment of `g1` and `g2` should ensure this property.

This paper attempts to show the advantages (and disadvantages) of using a temporal logic to reason about timed parallelism by presenting examples which illustrate how properties of such systems can be derived. This work is just one part of integrating formal hardware verification into the simulation approach to hardware design. The whole framework described in the first section has to be based upon a formal interpretation of the HDL in question, as is given in this paper. Work is underway to prove that the semantics presented here are consistent with a simplification of a Verilog simulator. Also, some techniques enabling the interpretation of a specification language as hardware have been developed. Hopefully, all this work will eventually tie up into a demonstration of how standard industry techniques can be reinforced by formal methods to provide a more flexible and robust set of tools.

References

1. E. Börger, U. Glässer, and W. Müller. Formal definition of an abstract VHDL '93 simulator by EA-machines. In C. Delgado Kloos and P.T. Breuer, editors, *Formal Semantics for VHDL*. Kluwer Academic Press Boston/London/Dordrecht, 1995.
2. P.T. Breuer, L. Sánchez, and C. Delgado Kloos. Clean formal semantics for VHDL. In *European Design and Test Conference, Paris*. IEEE Computer Society Press, 1993.
3. Albert Camilleri. Simulating hardware specifications within a theorem proving environment. *International Journal of Computer Aided VLSI design*, (2):315–337, 1990.
4. K.C. Davis. A denotational definition of the VHDL simulation kernel. In P. Agnew, L. Claesen, and R. Camposano, editors, *Proceedings of the 11th IFIP WG 10.2 International Conference on Computer Hardware Description Languages and their applications CHDL '93*, 1993.
5. Ivan V. Filippenko. VHDL verification in the State Delta Verification System (SDVS). In P.A. Subrahmanyam, editor, *Proceedings of the 1991 International Workshop on Formal Methods in VLSI design, Berlin*. Springer-Verlag, 1991.
6. Mike Gordon. The semantic challenge of Verilog HDL. In *Proceedings of the tenth annual IEEE symposium on Logic in Computer Science (LICS '95) San Diego, California*, June 1995.
7. Jifeng He and S.M. Brien. Z description of duration calculus. Technical report, Oxford University Computing Laboratory (PRG), 1993.
8. Jifeng He and Ernst-Rüdiger Olderog. From real-time specification to clocked circuit. ProCoS document, Department of Computer Science, Technical University of Denmark, DK-2800, Lyngby, Denmark, 1994.
9. Carlo Delgado Kloos and Peter T. Breuer. *Formal Semantics for VHDL*. Number 307 in The Kluwer International Series in Engineering and Computer Science. Kluwer Academic Publishers, 1995.
10. Xu Qiwen. Semantics and verification of extended phase transition systems in duration calculus. Research UNU/IIST Report No. 72, The United Nations University, International Institute for Software Technology, P.O. Box 3058, Macau, June 1996.
11. John Peter Van Tassel. A formalization of the VHDL simulation cycle. Technical Report 249, University of Cambridge Computer Laboratory, March 1992.
12. Chaochen Zhou. Duration calculi: An overview. In Dines Bjørner, Manfred Broy, and Igor V. Pottosin, editors, *Formal Methods in Programming and their Applications*, LNCS 735. Springer-Verlag, 1993.
13. Chaochen Zhou and Michael R. Hansen. Chopping a point. Technical report, Department of Information Technology, Technical University of Denmark, March 1996.
14. Chaochen Zhou, C.A.R. Hoare, and Anders P. Ravn. A calculus of durations. *Information Processing Letters*, 40(5):269–276, 1991.