PROSPER
ESPRIT LTR Project 26241

`http://www.dcs.gla.ac.uk/prosper`

Deliverable D2.1c

Semantics of the Intermediate Language IL

Version 1.5

Mike Gordon
Thomas Kropf
Dirk Hoffmann

Date released: February 12, 1999
Last modified: February 14, 1999

Mike Gordon
University of Cambridge Computer Laboratory
New Museums Site
Pembroke Street
Cambridge CB2 3QG
United Kingdom

work: (+44) 1223 334627
fax: (+44) 1223 334678
email: `mjcg@cl.cam.ac.uk`
web: `http://www.cl.cam.ac.uk/users/mjcg`

Thomas Kropf
University of Karlsruhe
Institute for Computer Design and Fault Tolerance
D-76128 Karlsruhe
Germany

work: (+49) 721 608 6326
fax: (+49) 721 608 3962
email: `kropf@ira.uka.de`
web: `http://goethe.ira.uka.de/~kropf`

Dirk Hoffmann
University of Karlsruhe
Institute for Computer Design and Fault Tolerance
D-76128 Karlsruhe
Germany

work: (+49) 721 608-7640
fax: (+49) 721 608-3962
email: `hoff@ira.uka.de`
web: `http://goethe.ira.uka.de/~hoff`

February 14, 1999

Contents

Preface to Version 1.1

At the end of December 1998, I started working on an explicit language, derived from SMV but minus some features and plus a generalised type system. This was the plan discussed at the November Prosper workshop and in December with Thomas Kropf in California.

However, as I wrote the syntax and semantics of IL down I found myself worrying about trivia like what the keywords should be called, whether they should be in upper or lower case, whether ";" should be a separator or teminator etc.

To avoid these trivial problems, I decided to switch to the approach used by Ian Page for his Handel HDL at Oxford: no concrete syntax, but abstract syntax as an ML datatype. This fitted nicely with Dirk Hoffman's ML abstract syntax datatype for SMV. Handel has a couple of concrete syntaxes (C-like for industry, ML-like for academics), but these are regarded as just user-interfaces. However, soon after starting on the IL abstract syntax datatype, I realised that I was prematurely fixing details that would need to evolve. In particular, I have little idea what extra stuff VHDL will require in addition to what's needed for Verilog.

Since D2.1c is specified to be a semantics, I felt justified in taking a final leap and eliminated abstract syntax as well as concrete syntax! The semantics of IL is presented as a set of semantic constructs, which are explicitly expected to evolve. There is a syntax for writing down semantic expressions (HOL term notation) so it isn't the case that by not giving IL an explicit syntax one is precluding writing and manipulating IL specifications. Furthermore, I think the approach is 'modern' in that it is in harmony with the SGML-XML philosophy of separating content (semantics) from presentation (syntax).

D2.1c Version 1.1 differs from Version 1.0β by incorporating suggestions, corrections and expository improvements. In particular, the treatment and notation for assignments and the representation of edges (`posedge`, `negedge`) has been modified. An additional section has been added giving examples illustrating how IL can be used to represent the semantics of synthesizable Verilog [32, 30, 11].

Mike Gordon
February 1, 1999

Preface to Version 1.3

In Version 1.3, the report has been extended with an introduction-section giving a brief overview over the formalization of HDLs. Besides mentioning general remarks about how HDLs can be formalized using logic, a brief summary is given about older research in this area.

Moreover, the report has been extended with a section about synthesizable VHDL. VHDL-processes are examined in more detail and the IL semantics of several VHDL programs is explained by giving examples.

Dirk Hoffmann
February 9, 1999

Preface to Version 1.5

Version 1.5 reorganises the treatment of assignments (by putting the various kinds into a single section). Also, the event control syntax has been replaced with one less like Verilog.

Old notation	**New notation**
`@(posedge clk)`	`rise clk ->`
`@(negedge clk)`	`fall clk ->`
`@(clk)`	`change clk ->`
`@(x or y or z)`	`change(x,y,z) ->`

This notation is still experimental.

Mike Gordon
February 12, 1999

1 Introduction

The overall aim of circuit design is the development of correct circuits and systems. This assumes the existence of a specification of the intended system behaviour. Having created a circuit implementation, methods of hardware verification may then be used to formally prove the correctness of the implementation with respect to the formal specification [16]. Hardware proofs are usually performed in logics. However, the standardized means of writing down hardware descriptions are languages such as VHDL [1] or Verilog [30]. This hinders the design of correct circuits in two ways: VHDL and Verilog are known to have imprecise semantics and there is a major gap between such HDLs as design languages and logics as formal systems for performing proofs in. To be able to carry out correctness proofs of VHDL or Verilog descriptions, a formal semantics for both languages has to be developed.

1.1 Methods for formalizing the semantics of HDL's

Two ways of formalizing HDLs are:

- *Shallow embedding.* A compiler is written to translate the given HDL program into a logic formula. The semantics of the HDL is coded only in the compiler.

- *Deep embedding.* All constructs of the HDL are defined within a logic by definitions of formulas for the (usually abstract) syntax and formulas for the semantics of these constructs.

There is a large semantic gap between HDLs and logic, i.e. single HDL constructs have large and complex semantics, resulting in large and complicated logic formulas. Thus, most works (see table 1) have reduced the complexity by using:

- the less secure, but simpler, shallow-embedding method;

- a subset of the HDL with simplified semantics;

- expressive logics (e.g. HOL, LAMBDA, OBJ3, Boyer-Moore);

- an intermediate formalisation language between the logic and the HDL.

Language	reference	subset	used logic	embedding
ART	[22]	complete	FOL	s
CASCADE	[27]	subset	Boyer-Moore	s
PML	[7]	complete	HOL	d
SILAGE	[3]	subset	HOL	d
SMAX	[8]	complete	FOL	d
STRICT	[20]	?	Boyer-Moore	s
ELLA	[4]	subset	HOL	s
ELLA	[10]	subset	LAMBDA	d
VHDL	[33]	subset	HOL	s
VHDL	[28]	subset	Boyer-Moore	s
VHDL	[34]	subset	LAMBDA	s
VHDL	[2]	subset	Boyer-Moore	s
VHDL	[5]	subset	temporal	s
VHDL	[21]	subset	temporal	s
VHDL	[31]	subset	HOL	d
VHDL	[26]	full	net-based	s
VHDL	[9]	subset	Prolog	d
VHDL	[6]	?	net-based	s
VHDL	[29]	subset	Boyer-Moore	d

Table 1: Classification of approaches for HDLs in formal hardware verification.

2 Design Goals

The intermediate language IL is intended to supply a common semantic representation for synthesisable subsets of Verilog and VHDL. It should be easy to translate from HDL into IL and from IL into the formal notations used for model checking and theorem proving.

This document describes the semantic concepts underlying IL. It is expected that IL will evolve during the lifetime of Prosper, so fixing on a concrete syntax now would be premature. Instead, a preliminary collection of semantic constructs is presented. These are motivated by consideration of the synthesisable subsets of Verilog and VHDL and are designed to interface cleanly with SMV. Future detailed work on HDL semantics may reveal the need to extend the IL semantics presented here. Once a robust set of constructs has been identified, then a particular concrete syntax for IL will be considered. However, it may turn out that a concrete syntax for IL is never needed, since designs can be represented directly as higher order logic terms. One possibility under consideration is to define IL via an XML DTD [14].

IL is intended for specifying sequential Mealy-style machines. Machines in IL can be quite general: e.g. partially specified, non-deterministic or infinite state. An important subclass are those machines that correspond to finite state transition relations. This is the subset that can be translated to SMV.

The semantics of a system is modelled as the set (represented as a predicate) of permitted *traces* of the inputs, states and outputs. This is a common approach used in theorem proving [13]. Model checking [24] takes as input a transition relation, often represented by a formula involving primed (next state) and unprimed (current state) variables. Although it is in general impossible to extract a generating finite transition relation from a set of traces, it is shown in Section 5 how a transition relation can be extracted from some combinations of IL semantic constructs. Such combinations of constructs constitute the subset of IL that supports model checking.

3 Semantic approach

The various semantic constructs of IL will be defined in the HOL logic [15].

Informal mnemonic notations for the constructs will be given. This should not be thought of as an official concrete syntax for IL. A more formal syntax, expressed as an XML DTD, or otherwise, may be developed later.

A specification consists of a formula containing variables ranging over traces. Traces have logical type $num \rightarrow \sigma$, where σ is a type representing data (bits, bytes, words etc.) and the type num of natural numbers represents position or *time* in the trace. The interpretation of the position depends on the level of temporal abstraction. It can be the number of clock ticks (e.g. rising edges) or a finer grain simulation time, as in HDL event simulators.

Variables are used to represent inputs and outputs. For example, a device with input i and output o, such that o carries the value input on i delayed by one time unit, is defined by the formula:

$$\forall t.\ o(t{+}1) \ = \ i(t)$$

which is written in IL as o `:=` i. Here the time $t : num$ might represent the number of clock ticks since initialization.

At a finer grain of time, suppose q gets the value of d whenever there is a rising edge on clk, and retains its previous value if there is no rising edge (i.e. a Dtype). Here t represents HDL simulation time. The formula relating the traces of d, clk and q is:

$$\forall t.\ q(t{+}1) \ = \ \texttt{if}\ \neg clk(t) \wedge clk(t{+}1)\ \texttt{then}\ d(t)\ \texttt{else}\ q(t)$$

In IL this is written as: `rise` clk `->` q `:=` d.

Finally, consider an unusual kind of Dtype that latches on the rising edge of clk if some condition P holds, but latches on the falling edge if P doesn't hold.

$$\begin{array}{lll} \forall t.\ (P(t) & \Rightarrow\ q(t{+}1) \ = \ \texttt{if}\ \neg clk(t) \wedge clk(t{+}1)\ \texttt{then}\ d(t)\ \texttt{else}\ q(t)) \\ \quad\ \ \wedge & \\ \quad\ \ (\neg P(t) & \Rightarrow\ q(t{+}1) \ = \ \texttt{if}\ clk(t) \wedge \neg clk(t{+}1)\ \texttt{then}\ d(t)\ \texttt{else}\ q(t)) \end{array}$$

In IL, this is written:

$$\begin{array}{ll} P & \Rightarrow\ \texttt{rise}\ clk\ \texttt{->}\ q\ \texttt{:=}\ d; \\ \neg P & \Rightarrow\ \texttt{fall}\ clk\ \texttt{->}\ q\ \texttt{:=}\ d \end{array}$$

3.1 Local variables

A machine may have some internal variables that are local to the specification.

For example, in:

w `:=` i`;` o `:=` w

the variable w might be considered local, the whole system just having input i and output o. The standard logical representation of hiding is existential quantification, thus in IL:

`local` $v_1 \ \cdots \ v_n$. B

means:

$\exists v_1 \ \cdots \ v_n. \ B$

3.2 Modules

Modules represent named hardware components. Each module has a *name*, a number of *parameters* and a *body*.

`module` <*name*>(<*parameter*>, ..., <*parameter*>)
<*body*>;
`end`

A module corresponds to an equation defining its name. Thus:

`module` M $(V_1, \ldots, V_m)$ B `end`

corresponds to the equation:

$$M \ = \ \lambda(V_1, \ldots, V_m). \ B$$

where V_1, ..., V_m are variables and B is a boolean term whose free variables are included amongst the parameters (this is necessary if the equation in HOL corresponding to a module is to be a valid definition).

The Module name M can either be declared as a constants or, in the case of submodules, it can be a variables. See Section 4.4. The logical type of a module name is $\sigma_1 \times \cdots \times \sigma_n \rightarrow bool$, where σ_i is the type of the ith parameter V_i.

Parameters will normally either be numbers representing sizes like bit-widths, or correspond to inputs or outputs. These two kinds of parameters are not distinguished.

Typically, the body is an existentially quantified conjunction of submodule definitions and variable initialization (Section 3.5) and assignments (Section 3.3). A semicolon will sometimes be used as a synonym for conjunction. Thus:

$$B_1; \ldots ; B_n$$

means

$$B_1 \ \wedge \ \ldots \ \wedge \ B_n$$

Note that ‘;’ is a separator, not a terminator.

Function applications will be written in both the standard mathematical form $f(x)$ or in the bracketless λ-calculus form $f \ x$.

3.3 Assignments

IL has three kinds of assignments: zero-delay assignments, unit-delay assignments and event-controlled assignments.

3.3.1 Zero-delay assignments

A *zero-delay assignment* defines the current value of a variable in terms of other current values. Zero-delay assignments are just equations between traces.

$$V \ = \ E$$

Traces are functions, so this is an equation between functions (values of type $num{\rightarrow}\sigma$) and so is equivalent to:

$$\forall t. \ V \ t \ = \ E \ t$$

Zero-delay assignments to local variables can often be eliminated by replacing occurrences of their left hand side by their right hand side using the one-point rule $((\exists x. \ x = e \ \wedge \ P(x)) \ = \ P(e))$; see Section 4.5 for an example.

3.3.2 Unit-delay assignments

The IL operator `next` is defined by:

$$\forall t.\ \texttt{next}\ V\ t\ =\ V(t{+}1)$$

The V `:=` E abbreviates:

$$\texttt{next}(V)\ =\ E$$

V is a variable of type $num{\to}\sigma$ and E is a term having the same type. The occurrence of '=' here has type $(num{\to}\sigma){\to}(num{\to}\sigma){\to}bool$.
Thus $\texttt{next}(V)\ =\ E$ and V `:=` E are equivalent and both mean:

$$\forall t.\ V(t{+}1)\ =\ E\ t$$

Note that in IL the assignment V `:=` E corresponds to the SMV assignment $\texttt{next}(V)$ `:=` E. An SMV assignment V `:=` E has no delay, so is like a zero-delay assignment in IL (see Section 3.3.1). I hope this doesn't cause confusion.
Terms, like E, on the right hand side of IL assignments are built out of variables and 'pointwise lifted' constants and operators. See Section 3.4 for more details.

3.3.3 Event controlled assignments

An event controlled assignment

$$P\ \texttt{->}\ V\ \texttt{:=}\ E$$

abbreviates:

$$\texttt{next}(V)\ =\ P\ \texttt{?}\ E\ :\ V$$

which has semantics:

$$\forall t.\ V(t{+}1)\ =\ \texttt{if}\ P(t)\ \texttt{then}\ E(t)\ \texttt{else}\ V(t)$$

Common events are clock edges; a rising edge on `clk` is defined by:

$$\texttt{rise}\ clk\ t\ \ =\ \ \neg(clk\ t)\ \wedge\ clk(t{+}1)$$

then a positive edge-triggered Dtype register is modelled by:

$$\texttt{rise}\ clk\ \texttt{->}\ q\ \texttt{:=}\ d$$

More generally define:

$$
\begin{array}{lcl}
\texttt{rise}\ V\ t & = & \neg V(t) \wedge V(t{+}1) \\
\texttt{fall}\ V\ t & = & V(t) \wedge \neg V(t{+}1) \\
\texttt{change}\ V\ t & = & \neg(V(t) = V(t{+}1)) \\
\texttt{change}(V_1,\ldots,V_n)\ t & = & \neg(V_1(t) = V_1(t{+}1))\ \vee\ \ldots\ \vee\ \neg(V_n(t) = V_n(t{+}1))
\end{array}
$$

Several assignments may be controlled by a single event control:

P `->` $(v_1$ `:=` E_1 `;` $\ldots$ `;` V_n `:=` $E_n)$

is equivalent to:

P `->` v_1 `:=` E_1 `;` $\ldots$ `;` P `->` V_n `:=` E_n

3.4 Operators on traces

Constants, unary and binary data operators are lifted from values to traces in the obvious way. For notational economy, the same symbol will be used for a constant or operator and its lifted counterpart. It is hoped this overloading doesn't confuse the reader. Disambiguation is straightforward by considering types.

For example:

$$
\begin{array}{lcl}
1 & = & \lambda t.\ 1 \\
\neg & = & \lambda t.\ \lambda b.\ \neg(b\ t) \\
+ & = & \lambda t.\ \lambda m\ n.\ (m\ t) + (n\ t)
\end{array}
$$

where the occurrences of '1', '$\neg$' and '+' on the left of the equation are lifted and the corresponding occurrences on the right are not.

As a concrete example of this overloading consider:

X `:=` $\neg X$

The '$\neg$' here is lifted, so has type $(num \to bool) \to (num \to bool)$. The semantics of this assignment is:

$$\forall t.\ X(t{+}1) = \neg X(t)$$

where the '$\neg$' here is not lifted, so has type $bool \to bool$.

The particular repertoire of operators in IL is not yet fixed. It is expected that useful libraries of operators will arise during the duration of Prosper.

An example of a useful operator is the ternary conditional operator:

$P?X:Y \;=\; \lambda t.\ \texttt{if}\ P(t)\ \texttt{then}\ X(t)\ \texttt{else}\ Y(t)$

This notation is similar to that used by C, Verilog and SMV.

It is assumed $-\texttt{?}-:-$ associates to the right, so that:

$P?X:Q?Y:Z \;=\; P?X:(Q?Y:Z)$

The one-armed conditional $P?X$ satisfies:

$\forall t.\ P(t) \;\Rightarrow\; (P?X)t \;=\; X\ t$

When P is false the value of $P?X$ is arbitrary ('don't care' for synthesis).

Other more complex n-ary operators (like `cases` and `switch` of SMV) are easily defined semantically.

3.5 Variable initializations

The initial value is at time 0 and is specified by:

`init` $<\!\mathit{variable}\!> = <\!\mathit{expression}\!>$

The semantics is that the initialization `init` $V = e$ denotes: $V\ 0 = e$.

4 Examples

The first three examples illustrate structural hierarchy.

4.1 Del: a unit-delay

A unit-delay models a register at the abstract state machine level.

```
module Del (i, o)
 o := i
end
```

This is equivalent to the definition:

$\mathsf{Del} \;=\; \lambda(i, o).\ \forall t.\ o(t{+}1) = i\ t$

4.2 Del2: two unit-delays in series

The following module has a local definition of the unit-delay Del and also a local variable x.

```
module Del2 (i, o)
 local Del x.
 module Del (i, o)
  o = i
 end;
 Del(i, x);
 Del(x, o)
end
```

Note that the typographical convention used here is that italics are used for variable names (e.g. x and Del) and sans serif for constants (e.g. $\mathsf{Del2}$). A top-level module definition defines a constant, but local submodule names are variables (variables can be quantified; constants can't).

Unfolding definitions yields:

$$\begin{array}{lll} \mathsf{Del2} \;=\; \lambda(i,o).\ \exists Del\ x. & (Del \;=\; \lambda(i,o).\ \forall t.\ o(t{+}1) = i\ t) & \wedge \\ & Del(i,x) & \wedge \\ & Del(x,o) & \end{array}$$

Simplifying using the one-point rule yields:

$$\mathsf{Del2} \;=\; \lambda(i,o).\ \exists x.\ (\forall t.\ x(t{+}1) = i\ t)\ \wedge\ (\forall t.\ o(t{+}1) = x\ t)$$

Using $((\forall t.\ P\ t)\ \wedge\ (\forall t.\ Q\ t)) \;=\; (\forall t.\ P\ t\ \wedge\ Q\ t)$ yields:

$$\mathsf{Del2} \;=\; \lambda(i,o).\ \exists x.\ \forall t.\ x(t{+}1) = i\ t\ \wedge\ o(t{+}1) = x\ t$$

4.3 Del4: two Del2s in series

The next example illustrates how confusion between module parameters and local variables with the same name is taken care of automatically by the standard logical treatment of free and bound variables.

```
module Del4 (i, o)
 local Del2 x.
 module Del2 (i, o)
  local Del x.
  module Del (i, o)
   o := i
  end;
  Del(i, x);
  Del(x, o)
 end
 Del2(i, x);
 Del2(x, o)
end
```

Expanding definitions and using the equation derived above for $Del2$ yields:

$$
\begin{array}{rcll}
\mathsf{Del4} & = & \lambda(i,o). & \\
 & & \exists Del2\ x. & \\
 & & \quad (Del2 = \lambda(i,o).\ \exists x.\ \forall t.\ x(t{+}1) = i\ t\ \wedge\ o(t{+}1) = x\ t) & \wedge \\
 & & \quad Del2(i,x) & \wedge \\
 & & \quad Del2(x,o) &
\end{array}
$$

Simplifying using the one-point rule, and renaming the existentially quantified bound variables to avoid capture yields:

$$
\begin{array}{rcll}
\mathsf{Del4} & = & \lambda(i,o).\ \exists x. & (\exists x_1.\ \forall t.\ x_1(t{+}1) = i\ t\ \wedge\ x(t{+}1) = x_1\ t)\ \wedge \\
 & & & (\exists x_2.\ \forall t.\ x_2(t{+}1) = x\ t\ \wedge\ o(t{+}1) = x_2\ t)
\end{array}
$$

Using $((\exists x_1.\ E_1)\ \wedge\ E_2)\ =\ (\exists x_1.\ E_1\ \wedge\ E_2)$ (which holds if x_1 desn't occur free in E_2) and $(E_1\ \wedge\ (\exists x_2.\ E_2))\ =\ (\exists x_2.\ E_1\ \wedge\ E_2)$ (which holds if x_2 desn't occur free in E_1) yields:

$$
\begin{array}{rcll}
\mathsf{Del4} & = & \lambda(i,o).\ \exists x\ x_1\ x_2. & (\forall t.\ x_1(t{+}1) = i\ t\ \wedge\ x(t{+}1) = x_1\ t)\ \wedge \\
 & & & (\forall t.\ x_2(t{+}1) = x\ t\ \wedge\ o(t{+}1) = x_2\ t)
\end{array}
$$

Moving $\forall$ up through $\wedge$ yields:

$$
\begin{array}{rcll}
\mathsf{Del4} & = & \lambda(i,o).\ \exists x\ x_1\ x_2.\ \forall t. & x_1(t{+}1) = i\ t\ \wedge\ x(t{+}1) = x_1\ t\ \wedge \\
 & & & x_2(t{+}1) = x\ t\ \wedge\ o(t{+}1) = x_2\ t
\end{array}
$$

which is equivalent to the flat module definition:

```
module Del4 (i,o)
 local x x1 x2.
 x1 := i;
 x  := x1;
 x2 := x;
 o  := x2
end
```

It is clear from this that a transition relation corresponding to Del4 is:

$$\lambda((i,o,x,x_1,x_2),(i',o',x',x_1',x_2')) \;=\; x_1' = i \;\wedge\; x' = x_1 \;\wedge\; x_2' = x \;\wedge\; o' = x_2$$

4.4 Local variables versus global constants

The previous definition of Del4 had a local module *Del*2, which in turn had a local module *Del*. Local modules are existentially quantified predicate variables. An alternative style, appropriate for general components, is to extend the theory by defining modules as constants.

With this approach one would make constant definitions:

```
module Del (i,o)
 o := i
end

module Del2 (i,o)
 local x.
 Del(i,x);
 Del(x,o)
end

module Del4 (i,o)
 local x.
 Del2(i,x);
 Del2(x,o)
end
```

Whether to declare modules as local variables or as global constants is a matter of style. Clearly with both approaches one ends up with the same semantics of the top-level module, namely:

$$\mathsf{Del4} \;=\; \lambda(i,o).\ \exists x\ x_1\ x_2.\ \forall t.\ \ \begin{array}{l} x_1(t{+}1) = i\ t\ \wedge\ x(t{+}1) = x_1\ t\ \wedge \\ x_2(t{+}1) = x\ t\ \wedge\ o(t{+}1) = x_2\ t \end{array}$$

4.5 Example with initialisation and combinational logic

Consider a version of Del4 in which the unit delay Del is initialized to v_0 at time 0 and there is a combinational function f inserted between the two Del2s in series:

```
module Del (i, o)
 init o = v0
 o := i
end

module Del2 (i, o)
 local x.
 Del(i, x);
 Del(x, o)
end

module Del4 (i, o)
 local x1 x2.
 Del2(i, x1);
 x2 = f(x1);
 Del2(x2, o)
end
```

With the added initialisation, the semantics of Del becomes:

$$\mathsf{Del} \;=\; \lambda(i,o).\ o(0) = \mathsf{v}_0\ \wedge\ \forall t.\ o(t{+}1) = i\ t$$

the semantics of Del2 becomes:

$$\mathsf{Del2} \;=\; \lambda(i,o).\ \exists x.\ \begin{array}{l} x(0) = \mathsf{v}_0\ \wedge\ o(0) = \mathsf{v}_0\ \wedge \\ \quad \forall t.\ x(t{+}1) = i(t)\ \wedge\ o(t{+}1) = x(t) \end{array}$$

and the semantics of Del4 becomes:

$$\mathsf{Del4} \;=\; \lambda(i,o).\ \exists x_1\ x_2.\ \begin{array}{ll} (\exists x_3.\ x_3(0) = \mathsf{v}_0\ \wedge\ x_1(0) = \mathsf{v}_0\ \wedge & \\ \quad \forall t.\ x_3(t{+}1) = i\ t\ \wedge\ x_1(t{+}1) = x_3\ t) & \wedge \\ (\forall t.\ x_2\ t\ = \mathsf{f}(x_1\ t)) & \wedge \\ (\exists x_4.\ x_4(0) = \mathsf{v}_0\ \wedge\ o(0) = \mathsf{v}_0\ \wedge & \\ \quad \forall t.\ x_4(t{+}1) = x_2\ t\ \wedge\ o(t{+}1) = x_4\ t) & \end{array}$$

which simplifies by eliminating the equation for $x_2\ t$ to:

$$
\begin{array}{rcll}
\mathsf{Del4} & = & \lambda(i,o).\ \exists x_1. & (\exists x_3.\ x_3(0) = \mathsf{v}_0\ \wedge\ x_1(0) = \mathsf{v}_0\ \wedge \\
 & & & \quad \forall t.\ x_3(t{+}1) = i\ t\ \wedge\ x_1(t{+}1) = x_3\ t)\ \wedge \\
 & & & (\exists x_4.\ x_4(0) = \mathsf{v}_0\ \wedge\ o(0) = \mathsf{v}_0\ \wedge \\
 & & & \quad \forall t.\ x_4(t{+}1) = \mathsf{f}(x_1\ t)\ \wedge\ o(t{+}1) = x_4\ t)
\end{array}
$$

and then (details left to the reader) to:

$$
\begin{array}{rcll}
\mathsf{Del4} & = & \lambda(i,o).\ \exists x_1\ x_3\ x_4. & x_3(0) = \mathsf{v}_0\ \wedge\ x_1(0) = \mathsf{v}_0\ \wedge \\
 & & & x_4(0) = \mathsf{v}_0\ \wedge\ o(0) = \mathsf{v}_0\ \wedge \\
 & & & \forall t.\ x_3(t{+}1) = i\ t\ \wedge\ x_1(t{+}1) = x_3\ t)\ \wedge \\
 & & & \quad x_4(t{+}1) = \mathsf{f}(x_1\ t)\ \wedge\ o(t{+}1) = x_4\ t
\end{array}
$$

which is equivalent to the flat module definition:

```
module Del4 (i,o)
 local x1 x3 x4.
  init x3 = v0;
  init x1 = v0;
  init x3 = v0;
  init o  = v0;
  x3  := i;
  x1  := x3;
  x4  := f x1;
  o   := x4
end
```

The state transition system corresponding to this has initial state specified by:

$$x_1 = \mathsf{v}_0\ \wedge\ x_3 = \mathsf{v}_0\ \wedge\ x_4 = \mathsf{v}_0\ \wedge\ o = \mathsf{v}_0$$

and transition relation:

$$
\begin{array}{l}
\lambda((i,o,x_1,x_3,x_4),(i',o',x',x_1',x_3',x_4'))\ = \\
\quad x_3' = i\ \wedge\ x_1' = x_3\ \wedge\ x_4' = \mathsf{f}\ x_1\ \wedge\ o' = x_4
\end{array}
$$

4.6 A simple state-machine

The example below is from 1.2.4 of Kenneth McMillan's book entitled *The SMV Language* [23].

The variable $state$ can take one of three values: idle, $\mathsf{cyc1}$, $\mathsf{cyc2}$. To support this, define a new type:

$$state \;=\; \mathsf{idle} \;\mid\; \mathsf{cyc1} \;\mid\; \mathsf{cyc2}$$

The variables $start$ and $done$ are boolean (i.e. of type $bool$).

$$
\begin{array}{l}
\texttt{module}\ \mathsf{StateMachine}\ (start, done) \\
\ \texttt{local}\ state. \\
\ \texttt{init}\ state = \mathsf{idle}; \\
\ state\ \ \texttt{:=}\ \ \begin{array}[t]{l}(state = \mathsf{idle})\ \ \texttt{?}\ (start\texttt{?}\mathsf{cyc1}\texttt{:}\mathsf{cyc2})\ \texttt{:} \\ (state = \mathsf{cyc1})\ \texttt{?}\ \mathsf{cyc2}\ \texttt{:} \\ (state = \mathsf{cyc2})\ \texttt{?}\ \mathsf{idle};\end{array} \\
\ done = (state = \mathsf{cyc2}) \\
\texttt{end}
\end{array}
$$

Exercise. Write down the semantics of StateMachine and the derived state transition system (see Section 5 below).

4.7 Dreg: a clocked Dtype register

A standard positive edge triggered Dtype register that is initilised to store F (i.e. falsity or $\mathtt{0}$) is defined by:

$$
\begin{array}{l}
\texttt{module}\ \mathsf{Dreg}\ (clk, d, q, \overline{q}) \\
\ \texttt{init}\ q = \mathsf{F}; \\
\ \texttt{rise}\ clk\ \texttt{->}\ q\ \texttt{:=}\ d; \\
\ \overline{q} = \neg q \\
\texttt{end}
\end{array}
$$

This means:

$$
\begin{array}{lcl}
\mathsf{Dreg} & = & \lambda(clk, d, q, \overline{q}). \\
& & \quad \begin{array}{ll} q(0) = \mathsf{F} & \wedge \\ (\forall t.\ q(t{+}1)\ =\ \texttt{if}\ \texttt{rise}\ clk\ t\ \texttt{then}\ d\ t\ \texttt{else}\ q\ t) & \wedge \\ (\forall t.\ \overline{q}\ t\ =\ \neg(q\ t)) & \end{array}
\end{array}
$$

i.e.

$$
\begin{array}{lcl}
\mathsf{Dreg} & = & \lambda(clk, d, q, \overline{q}). \\
& & \quad \begin{array}{ll} q(0) = \mathsf{F} & \wedge \\ (\forall t.\ q(t{+}1)\ =\ \texttt{if}\ \neg(clk\ t) \wedge clk(t{+}1)\ \texttt{then}\ d\ t\ \texttt{else}\ q\ t) & \wedge \\ (\forall t.\ \overline{q}\ t\ =\ \neg(q\ t)) & \end{array}
\end{array}
$$

Note that the zero-delay assignment to $\overline{q}$ cannot be eliminated, since $\overline{q}$ is not a local variable, but is an output.

The state transition system corresponding to this has initial state specified by:

$$q\ 0 \ = \ \mathsf{F} \ \wedge \ \overline{q}\ 0 \ = \ \mathsf{T}$$

and transition relation:

$$\begin{array}{l} \lambda((clk, d, q, \overline{q}), (clk', d', q', \overline{q}')) \ = \\ \quad (q' \ = \ \texttt{if}\ \neg clk \wedge clk'\ \texttt{then}\ d\ \texttt{else}\ q) \ \wedge \ (\overline{q} \ = \ \neg q) \ \wedge \ (\overline{q}' \ = \ \neg q') \end{array}$$

5 Extracting transition relations

As has been illustrated via the examples, many modules can be flattened by logical manipulation into a form:

$$\begin{array}{l} \texttt{module}\ <\!name\!>\ (<\!parameters\!>) \\ \ \texttt{init}\ <\!initialisation\!>; \\ \ \texttt{local}\ <\!variables\!>. \\ \ <\!unit\!-\!delay\ \ assignments\!>; \\ \ <\!zero\!-\!delay\ \ assignments\!> \\ \texttt{end} \end{array}$$

whose semantics is, roughly speaking:

$$\begin{array}{lll} <\!name\!> \ = \ \lambda(<\!parameters\!>). & \exists <\!variables\!>. & \\ & <\!initialisation\!> \wedge & \\ & \forall t.\ <\!unit\!-\!delay\ \ assignments\!> \wedge & \\ & \quad\ <\!zero\!-\!delay\ \ assignments\!> & \end{array}$$

If all the parameters and variables have finite types, then a formula like this determines a state transition system. Consider:

$$\begin{array}{l} \texttt{module}\ \mathsf{M}\ (p_1, \ldots, p_m) \\ \ \texttt{local}\ l_1\ \cdots\ l_n. \\ \ \texttt{init}\ v_1 = e_1;\ \cdots\ ;\ \texttt{init}\ v_r = e_r; \\ \ v_1\ \texttt{:=}\ E_1;\ \ldots\ ;\ v_p\ \texttt{:=}\ E_p; \\ \ v_q = E_q;\ \ldots\ ;\ v_r = E_r \\ \texttt{end} \end{array}$$

In this general form, the variables assigned to $v_1, v_2, \ldots$ are a subset of the union of the parameters $p_1, \ldots, p_m$ and the local variables $l_1, \ldots, l_n$. The initialisation expressions $e_1, \ldots, e_r$ are value expressions not containing parameters or locals. The assignment right hand sides $E_1, \ldots, E_p$ are trace-valued expressions that may contain occurrences of the parameters and locals.

Let $E'_q, \ldots, E'_r$ be the expressions obtained from $E_q, \ldots, E_r$ by replacing occurrences of $v_1, \ldots, v_r$ by the primed variables $v'_1, \ldots, v'_r$, respectively. The module above corresponds to the state transition system with initial set of states defined by:

$$v_1\ 0 \ = \ e_1 \ \wedge \ \cdots \ \wedge \ v_r\ 0 \ = \ e_r$$

and transition relation:

$$\begin{array}{l}
\lambda((v_1, \ldots, v_r), (v'_1, \ldots, v'_r)) \ = \\
\quad v'_1 = E_1 \ \wedge \ \cdots \ \wedge \ v'_p = E_p \ \wedge \\
\quad v_q = E_q \ \wedge \ \cdots \ \wedge \ v_r = E_r \ \wedge \\
\quad v'_q = E'_q \ \wedge \ \cdots \ \wedge \ v'_r = E'_r
\end{array}$$

Note that the unit-delay assignments specify the next state (primed variables) in terms of the current state (unprimed variables), but the zero-delay assignments specify the values of variables in both the current and next states.

Note that the Dtype register in Section 4.7 is not of this form, but still determines a transition system. Thus the method of extracting transition systems from IL semantic representation described here can be extended to cover a wider class of examples. It is expected that such extensions will be developed during the Prosper project.

Exercise. Consider how to generalise the method of extracting transition systems given here so that the transition system for Dreg of Section 4.7 can be automatically derived.

6 Different timescales

An important feature of the semantics presented here is that it supports abstraction and refinement. Modules are predicates and thus abstraction/refinement can be represented by logical implication. In the context of theorem proving, this has been worked out in detail [18, 25]. Considerable current work aims to use abstraction as a basis for combining model checking and theorem proving. The framework described here is intended to facilitate this.

Recall from Section 4.1 the unit-delay example:

```
module Del (i, o)
 o := i
end
```

which is equivalent to:

$$\mathsf{Del} \;=\; \lambda(i,o).\ \forall t.\ o(t{+}1) = i\ t$$

The timescale of this example is the 'clock tick' scale – i.e. the value of the time variable t is the number of clock ticks since initialisation.

Consider now a version of Dreg without the $\overline{q}$ output:

```
module Dtype (clk, d, q)
 init q = F;
 rise clk -> q := d
end
```

which means:

$$\begin{array}{lll} \mathsf{Dtype} & = & \lambda(clk,d,q). \\ & & \quad q(0) = \mathsf{F} \wedge \\ & & \quad (\forall t.\ q(t{+}1) \;=\; \texttt{if}\ \neg(clk\ t) \wedge clk(t{+}1)\ \texttt{then}\ d\ t\ \texttt{else}\ q\ t) \end{array}$$

The timescale here is much finer. It is similar to the timescale used by event simulators for HDLs. The more abstract timescale used for Del is the timescale used in cycle-based simulation.

The relation between Del and Dtype is one of *temporal abstraction*:

$$\begin{array}{l} \forall clk.\ \mathsf{Inf}(\mathsf{Rise}\ clk) \;\Rightarrow \\ \qquad \forall d\ q.\ \mathsf{Dtype}(clk,d,q) \;\Rightarrow\; \mathsf{Del}(d\ \mathsf{when}\ (\mathsf{Rise}\ clk), q\ \mathsf{when}\ (\mathsf{Rise}\ clk)) \end{array}$$

The various operators used here are explained in Melham's book [25, p. 104]; they are similar to ones found in LUSTRE [17].

7 Semantics of synthesizable Verilog

In this section, a semantics for a reasonably large subset of synthesizable Verilog is outlined via examples.

The semantics reflects a synthesis interpretation of Verilog: HDL text is viewed as specifying a collection of state machines (one for each Verilog `always`-block). This should be contrasted with a simulation interpretation, in which HDL is interpreted by an event scheduling algorithm. It is a hard unsolved problem to ensure that the synthesis and simulations views coincide. Solving this is important if one is to have complete confidence that synthesised hardware will have properties predicted by simulation. This problem is not addressed here (but a start has been made elsewhere [12]).

The semantics of Verilog splits into three largely orthogonal aspects: expression evaluation, statement interpretation and module elaboration.

7.1 Expression evaluation

Expression evaluation concerns the detailed semantics of the Verilog operators on the various datatypes. There are many subtleties arising out of the way operators can truncate or pad bitstrings to meet declared size specifications. Also, the values `x` and `z` are tricky and tend to be interpreted differently by simulation and synthesis. The value `x` is an undefined value for simulation and a don't care value for synthesis. Certain occurrences of the value `z` cause tri-state logic to be synthesized, but it is not clear that this always corresponds to how simulators treat `z`. Verilog's expression evaluation is not considered here, but a detailed analysis can be found in a report by Myra Van Inwegen [19]. It is assumed that HOL datatypes corresponding to Verilog types will be defined, together with appropriate operators on the values.

7.2 Statement interpretation

The semantics of statements in IL will be illustrated via examples. These are adapted from an unpublished report [12], where a general procedure for interpreting statements is given.

Statements occur in Verilog modules as `always`-blocks of the form `always` $\mathcal{S}$. The following examples show the IL semantics of a number of such `always`-blocks. When there is more than one event control in a statement, a local program counter `pc` is used to encode which event control is active. Other ways of encoding control state are possible (e.g. one-hot).

Example 1

The example below sets `a` to `0` on the first edge and then sets `b` to `a` on the second edge. Thereafter `a` and `b` are updated with `0` on each cycle.

```
always @(posedge clk) begin a=0; @(posedge clk) b=a; end
```

A semantics in IL is:

```
local pc.
 init pc = 0;
 pc = 0 ⇒ rise clk -> (pc := 1; a := 0; b := b);
 pc = 1 ⇒ rise clk -> (pc := 0; a := a; b := a)
```

N.B.: do not confuse Verilog's semantics of ';' inside `begin`–`end` blocks (sequencing) from IL's use of ';' (conjunction).

Example 2

The following example is a state machine described in an implicit style. It is Example 8-16 from the Synopsys HDL Compiler for Verilog Reference Manual [30].

```
always
 begin
  @(posedge clk) total = data;
  @(posedge clk) total = total + data;
  @(posedge clk) total = total + data;
 end
```

A semantics in IL:

```
local pc.
 init pc = 0;
 pc = 0 ⇒ rise clk -> (pc := 1; total := data);
 pc = 1 ⇒ rise clk -> (pc := 2; total := total+data);
 pc = 2 ⇒ rise clk -> (pc := 0; total := total+data)
```

Example 3

An explicit style of description of the machine in Example 2 is given next. This is Example 8-17 from the Synopsys HDL Compiler for Verilog Reference Manual [30].

```
always
 @(posedge clk)
 begin
  case (state)
  0:  begin total = data;
            state = 1;
      end
  1:  begin total = total + data;
            state = 2;
      end
  default:
      begin total = total + data;
            state = 0;
      end
  endcase
 end
```

Semantics in IL:

```
rise clk -> (total := (state = 0) ? data : total+data;
             state := (state = 0) ? 1 : (state = 1) ? 2 : 0)
```

The explicit states style of state machine eliminates the need for a program counter.

Example 4

Another example not requiring a program counter is:

```
always @(posedge clk)
          if (p) begin a=b;  b=a;  end
          else   begin a<=b; b<=a; end
```

Here `<=` is Verilog's non-blocking assignment. The semantics is:

```
rise clk -> (a := b;
             b := p ? b : a)
```

Example 5

Asynchronous (combinational) always blocks also don't need program counters. For example:

```
always @(b or c) a = b + c
```

has semantics:

```
change(b,c) -> a := b+c
```

Whenever `b` and `c` change, `a` is updated, hence it follows (induction over time) that the equation $\forall t.\ \mathtt{a}(t) = \mathtt{b}(t) + \mathtt{c}(t)$ is an invariant.

Thus `change(b,c) -> a := b+c` is equivalent to the zero-delay assignment `a = b+c`.

However consider instead:

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

This has semantics:

```
change b -> a := b+c
```

Suppose `a` equals `b+c`. If `c` changes, then the event control `change b` is not triggered, so `a` won't be updated and may become different from `b+c`. Thus `a`'s value must be latched, so this does not define combinational logic. This illustrates the need for synthesizers to do latch inference (and/or warn about suspected incomplete sensitivity lists).

Example 6

Here is a combinational example that doesn't lead to any latch inference.

```
always
 @(a or b or c or d)
 begin
  f = a;
  if (b)
   begin
    if (c) f = d; else f = !d;
   end
 end
```

Semantics:

```
change(a,b,c,d) -> f := b ? (c ? d : ¬d) : a
```

Example 7

This example is unlikely to arise in practice:

```
  always if (p) begin
                   a=1;
                   @(posedge clk) b=2;
                   @(negedge clk) c=3;
                end
         else   begin
                   a=5;
                   @(clk) b=6;
                end
```

The semantics is:

```
local pc.
 init pc = p ? 1 : 3;
 init a  = p ? 1 : 5;
 pc = 1 ⇒ rise clk   -> (pc := 2;
                          c  := c;
                          a  := a;
                          b  := 2);
 pc = 2 ⇒ fall clk   -> (pc := p ? 1 : 3;
                          c  := 3;
                          a  := a;
                          b  := 2);
 pc = 3 ⇒ change clk -> (pc := p ? 1 : 3;
                          c  := 3;
                          a  := p ? 1 : 5;
                          b  := b)
```

Example 8

```
always @(posedge clk)
 begin
  X = Inp1; Y = X+Inp2;
  @(posedge clk)
  OUT = X+Y;
 end
```

has semantics:

```
local pc.
 init pc = 0;
 pc = 0 ⇒ rise clk -> (pc  = 1;
                       X   = Inp1;
                       Y   = Inp1+Inp2;
                       OUT = OUT);
 pc = 1 ⇒ rise clk -> (pc  = 0;
                       X   = X ;
                       Y   = Y;
                       OUT = X+Y)
```

Example 9

```
always @(posedge clk)
 begin
  if (Choose) X = In1; else X = In2;
  @(posedge clk)
  OUT = X + 1;
 end
```

has semantics:

```
local pc.
 init pc = 0;
 pc = 0 ⇒ rise clk -> (pc  = 1;
                       X   = Choose ? In1 : In2;
                       OUT = OUT);
 pc = 1 ⇒ rise clk -> (pc  = 0;
                       X   = X;
                       OUT = X + 1)
```

Example 10

```
always @(posedge clk)
 begin
  X = Inp1; Y = Inp2; OUT=X;
  while (Y > 1)
   begin
    @(posedge clk)
    OUT = OUT+X; Y = Y-1;
   end
 end
```

is a naive multiplier and has semantics:

```
local pc.
 init pc = 0;
 pc = 0 rise clk -> (pc  = Inp2 > 1 ? 1 : 0;
                     X   = Inp1;
                     OUT = Inp1;
                     Y   = Inp2);
 pc = 1 rise clk -> (pc  = Y-1 > 1 ? 1 : 0;
                     X   = X;
                     OUT = OUT+X;
                     Y   = Y-1)
```

A systematic treatment of a reasonably large class of Verilog statements is considered in an unpublished report [12], including block disables, `while`-statements, `for`-statements and `case`-statements. It is hoped to develop a more compositional semantic treatment than the one described there, but it does at least provide a reassuring fall-back position.

7.3 Module elaboration

The elaboration of Verilog modules will be handled using IL module constructs, as illustrated in Section 4.

A Verilog specification consists of a collection of modules. Modules cannot have local submodules, so the approach described in Section 4.4, in which each module defines a constant, is appropriate.

Modules can have local *functions*, *tasks*, *wires* and *registers*. Functions define abbreviations for combinational logic, whereas tasks are abbreviations for state machines. Wires are driven continuously, whereas registers hold persistent state which is modified by a programming-like assignment statement.

Because types are being glossed over here, type-related information will be ignored. This includes bit-widths, input/output/inout declarations and the distinction between wires and registers. Also, functions and tasks are given explicit parameter lists, rather than (as is the case in Verilog proper) parameters being declared inside function and task bodies using input/output declarations.

A typical Verilog module declaration will be taken to have the form shown below.

```
module <name> (<name>,...,<name>);
    ⋮
 function <name>(<name>,...,<name>);
   <statement>
 endfunction
    ⋮
 task <name>;(<name>,...,<name>)
   <statement>
 endfunction
    ⋮
 wire <name>;
    ⋮
 reg <name>;
    ⋮
 assign <name> = <expression>;
    ⋮
 always <statement>;
    ⋮
 <name> (<expression>,...,<expression>);
    ⋮
endmodule
```

The semantics of such a module is an IL module with the same name and parameters. The `wire` and `reg` declarations become `local` declarations.

Functions are converted to IL initialised local variables. For example:

```
function f(a,b,c,d);
 begin
  f = a;
  if (b)
   begin
    if (c) f = d; else f = !d;
   end
 end
```

generates:

```
local f.
 init f = λ(a,b,c,d). b ? (c ? d : ¬d) : a
```

Tasks are elaborated by replacing each invocation by the appropriate instance of the body (i.e. call-by-name).

Continuous assignments are converted to zero-delay assignments, i.e. equations, as described in Section 3.3.1.

`always`-blocks are converted to IL; see the examples in Section 7.2.

Examples

The examples in this section were partially done by hand and could contain errors. The first example has a continuous assignment, and illustrates a situation when simulation and synthesis can disagree.

```
module BadXL(clk, in, out);
   input clk, in;
   output out;
   reg    out, r;
   wire   w;
   assign w = r;
   always @ (posedge clk) begin
      r = in;
      out = w;
   end
endmodule
```

This has semantics:

```
module BadXL (clk,in,out);
 local r w.
  w = r;
  rise clk -> (r := in; out = w)
end
```

which is semantically equivalent to:

```
module BadXL (clk,in,out);
  rise clk -> out := in
end
```

An example using tasks is the following (unrealistic) device to naively compute the factorial function.

This uses a task `Mult` to perform multiplication by adding the first argument (`InOut`) to itself the second argument (`In`) number of times. The result is thus returned in the first argument.

```
 module Fact (Start,Inp,clk,FACT,BUSY);
  reg N;
  task Mult(InOut,In);
   reg X Y;
   always @(posedge clk)
    begin
     X = InOut; Y = In;
     while (Y > 1)
      begin
       @(posedge clk)
       InOut = InOut+X; Y = Y-1;
      end
    end
  endtask;
  always @(posedge clk)
   if (Start)
      begin N = Inp;
            FACT = 1; BUSY = 1;
            while (N>1)
             begin
              Mult(FACT,N);
              N = N-1;
             end
             BUSY = 0;
      end
 endmodule
```

This has semantics:

```
module Fact (Start,Inp,clk,FACT,BUSY);
 local pc X Y N.
  init pc = 0;
  pc = 0 ⇒ rise clk -> (pc   := Start ? (Inp>1?1:0) : 0;
                         X    := X;
                         FACT := Start ? 1 : FACT;
                         Y    := Y;
                         N    := Start ? Inp : N;
                         BUSY := Start ? (Inp>1?1:0) : BUSY);
  pc = 1 ⇒ rise clk -> (pc   := N>1 ? 2 : (N-1>1?1:0);
                         X    := FACT;
                         FACT := FACT;
                         Y    := N;
                         N    := N>1 ? N : N-1;
                         BUSY := N>1 ? BUSY : (N-1>1?BUSY:0));
  pc = 2 ⇒ rise clk -> (pc   := Y-1>1 ? 2 : (N-1>1?1:0)
                         X    := X;
                         FACT := FACT+X;
                         Y    := Y-1;
                         N    := Y-1>1 ? N : N-1;
                         BUSY := Y-1>1 ? BUSY : (N-1>1?BUSY:0))
end
```

The examples here are intended to illustrate how IL is used to give the synthesis semantics of Verilog. A more systematic presentation will be given as a later deliverable.

8 Semantics of synthesizable VHDL

8.1 VHDL processes

In VHDL, the primary elements to model a state based system are *processes.* The body of a process roughly consists of a *process declarative part* and a *sequential statement part.* Using a BNF style grammar, processes are defined as

process	::=	[process_label :] `process` [(*sensitivity_list*)]
		process_declarative_part
		`begin`
		process_statement_part
		`end process` [process_label];
sensitivity_list	::=	list of signal names

In the process delarative part, local items such as variables and constants can be defined.

The sequential statement part is a list of statements and is activated in response to changes in state. Processes that are activated at the same time run concurrently.

A process in VHDL is the counterpart to an `always` construct in Verilog. Roughly speaking, the Verilog program

```
always
 begin
  P;
 end
```

is equivalent to the VHDL program

```
p0 : process
 begin
  P;
 end process p0
```

While in Verilog, timing controls are always expressed with the `@` operator, VHDL provides two different ways to control time.

- Control by *sensitivity lists*
- Control by *wait statements*

In VHDL, a sensitivity list can be attached to each process and consists of one or more signal names. Whenever a change occurs in some signal belonging to the sensitivity list of a process p, the sequential statement part of p is executed. VHDL processes with no or empty sensitivity lists are treated in a special way: They are always executed and can only be suspended by a wait statement.

A process p with sensitivity list $(V_1, \ldots, V_n)$ has the same functionality as Verilog's

$$\texttt{always@ (}V_1 \texttt{ or } \ldots \texttt{ or } V_n\texttt{)}$$

construct, where $V_1, \ldots, V_n$ are restricted to be *change of value* event expressions, i.e., $V_1, \ldots, V_n$ specify the names of signals. Thus sensitivity lists in VHDL are less expressive as event expressions in Verilog. For example, it is not possible to express `@posedge` or `@negedge` timing controls.

Wait statements are the second possibility to express timing control in VHDL and are more expressive than sensitivity lists.

A wait statement is defined as

$$\begin{array}{rcl} \textit{wait\_statement} & ::= & \texttt{wait } [\textit{sensitivity\_clause}][\textit{condition\_clause}] \\ \textit{sensitivity\_clause} & ::= & \texttt{on } \textit{sensitivity\_list} \\ \textit{sensitivity\_list} & ::= & \text{list of signal names} \\ \textit{condition\_clause} & ::= & \texttt{until } \text{condition} \end{array}$$

Whenever a process p is executed and a wait statement is processed, p will be suspended. The sensitivity list of the wait statement specifies a set of signals to which the process is sensitive while being inactive.

Whenever a signal in the sensitivity list gets changes, the *condition clause* is evaluated. If the condition is true, the execution of the process is continued. Otherwise, the process resuspends.

Using conditional wait statements, all event controls of Verilog can be expressed. The event control `@posedge clk` can be written as

```
process
 begin
  wait on clk until clk = '0';
  wait on clk until clk = '1';
  ...
 end process;
```

The two wait statements can only be passed if the value of clock switches from zero to one. Another possibility to detect rising edges is

```
process (clk)
 begin
  wait on clk until clk = '1';
  ...
 end process;
```

Here, a sensitivity list is combined and a wait statement are combined. Unlike in the first program fragment, the sensitivity list is not empty and therefore the process is invoked if and only if a signal change occured on `clk`. The wait statement then forces the process to wait until `clk` rises.

The next possibility is probably the most common way for detecting edges (especially in synthesizable VHDL) and makes use of the predefined function `RISING_EDGE`:

```
process
 begin
  wait on clk until RISING_EDGE(clk);
  ...
 end process;
```

Similar to `@posedge clk`, the Verilog event control `@negedge clk` can be expressed as

```
process
 begin
  wait on clk until clk = '1';
  wait on clk until clk = '0';
  ...
 end process;
```

or

```
process (clk)
 begin
  wait on clk until clk = '0';
  ...
 end process;
```

or

```
process
 begin
  wait on clk until FALLING_EDGE;
  ...
 end process;
```

`RISING_EDGE`(s) and `FALLING_EDGE`(s) are predefined functions evaluating to true iff s is currently rising or falling, respectively.

In terms of expressiveness, wait statements in VHDL subsume a sensitivity lists. Each VHDL program of the form

```
p0 : process (x,y,z)
 begin
  P
 end process p0
```

can be rewritten as

```
p0 : process
 begin
  wait on (x,y,z) until ( x'EVENT or y'EVENT or z'EVENT);
  P
 end
```

`EVENT` is another predefined VHDL construct. `x'EVENT` is true if and only if an event (signal change) occured on `x`.

Using the `EVENT` construct, the functions `RISING_EDGE` and `FALLING_EDGE` can be expressed naturally as

$$\begin{aligned} \texttt{RISING\_EDGE}\ s &= s'\texttt{EVENT and}\ s = \text{'1'} \\ \texttt{FALLING\_EDGE}\ s &= s'\texttt{EVENT and}\ s = \text{'0'} \end{aligned}$$

In the synthesizable subset of VHDL, wait-statements can only be used in a very restricted way. Each process is only allowed to have one single wait-statement which has to occur as the first command in the sequential statement part. No local sensitivity lists are allowed for wait statements. Thus, edge detection must be realized with functions `RISING_EDGE` and `FALLING_EDGE`, or the corresponding counterparts based on the `EVENT` expression.

As a result, the expressiveness of the wait construct is cut down to roughly the same expressiveness as provided by Verilog's timing controls.

8.2 VHDL semantics by example

Similar to Section 7, a sematics for synthesizable VHDL is outlined by example.

Example 1

As an introductory example, let us first consider the following VHDL program:

```
signal a,b : bit;
signal pc  : bit := 0;
p0 : process
 begin
  wait on clk until RISING_EDGE clk;
  case pc of
   when 0 =>
    pc <= 1;
     a <= 0;
   when 1 =>
    pc <= 0;
     b <= a;
  end case;
 end process p0;
```

Expressing the *case*-statement as a nested construct of if-then-else operators $(X?Y : Z)$, the program has the IL sematics

```
local pc.
 init pc = 0;
 rise clk -> (pc = 0 ? (pc := 1; a := 0; b := b)
                     : (pc = 1 ? (pc := 0; a := a; b := a)
                                : 0))
```

Exploiting the equivalence

`rise` x `->` (A `?` B `:` C) `=` A `?` (`rise` x `->` B) `:` (`rise` x `->` C)

this is equivalent to

```
local pc.
 init pc = 0;
 pc = 0 ? rise clk -> (pc := 1; a := 0; b := b)
        : (pc = 1 ? rise clk -> (pc := 0; a := a; b := a)
                  : 0)
```

Using the rule

$$(A \ ? \ B \ : \ C) \ = \ (A \Rightarrow B \ ; \ \neg A \Rightarrow C))$$

for rewriting the *if-then-else* construct with *conjuction* (;) and *implication* (⇒), the program can be written as

```
local pc.
 init pc = 0;
 (pc = 0) ⇒ rise clk -> (pc := 1; a := 0; b := b);
 ¬(pc = 0) ⇒
     ( (pc = 1) ⇒ rise clk -> (pc := 0; a := a; b := a);
       ¬(pc = 1) ⇒ 0)
```

Since pc has type *bit*, we can conclude

```
¬(pc = 0)  =  (pc = 1)
¬(pc = 1)  =  (pc = 0)
```

and rewriting these equations result in

```
local pc.
 init pc = 0;
 pc = 0 ⇒ rise clk -> (pc := 1; a := 0; b := b);
 pc = 1 ⇒
   ((pc = 1) ⇒ rise clk -> (pc := 0; a := a; b := a);
    (pc = 0) ⇒ 0)
```

which is equivalent to

```
local pc.
 init pc = 0;
 pc = 0 ⇒ rise clk -> (pc := 1; a := 0; b := b);
 pc = 1 ⇒ rise clk -> (pc := 0; a := a; b := a)
```

This is exactly the result we got for the Verilog example

```
always @(posedge clk) begin a = 0; @(posedge clk) b = a; end
```

presented in Section 8.2. Thus, both programs are equivalent.

Example 2

The next example is the VHDL equivalent to Verilog-example 3:

```
signal pc : [0..2];
signal total, data : integer;

p0 : process
 begin
  wait until  RISING_EDGE(clk);
  case pc of
   when 0 =>
    pc <= 1;
    total <= data;
   when 1 =>
    pc <= 2;
    total <= total+data;
   when 2 =>
    pc <= 0;
    total <= total+data;
 end case;
end process p0;
```

which has semantics

```
local pc.
 rise clk ->
  (pc = 0 ⇒ (pc := 1; total := data);
   pc = 1 ⇒ (pc := 2; total := total+data);
   pc = 2 ⇒ (pc := 0; total := total+data))
```

or

```
local pc.
 pc = 0 ⇒ rise clk -> (pc := 1; total := data);
 pc = 1 ⇒ rise clk -> (pc := 2; total := total+data);
 pc = 2 ⇒ rise clk -> (pc := 0; total := total+data)
```

which is equivalent.

Example 3

```
signal a,b,c : integer;
p0 : process (b,c)
 begin
  a <= b+c;
 end process p0;
```

has semantics

```
@((edge b) or (edge c)) (a := b+c)
```

or

```
a = b+c
```

Here, a zero-delay assignment can be chosen since the sensitivity list covers all signals occuring on a right side of a signal assignment.

Example 4

```
signal a,b,c : integer;
p0 : process (b)
 begin
  a <= b+c;
 end process p0;
```

The only difference to Example 3 is the sensitivity list where signal `c` has been removed. Both programs are not equivalent as it was the case for the corresonding Verilog examples presented in Section 8.2.

The IL semantics is given by

```
@(edge b) (a := b+c)
```

We now give three examples of concurrent processes:

Example 5

```
signal a,b,c;

p0 : process
 begin
  wait on clk until RISING_EDGE(clk)
   a <= c;
 end process p0;

p1 : process
 begin
  wait on clk until FALLING_EDGE(clk)
   b <= a;
 end process p0;
```

This VHDL program assigns the value of `c` to `a` at each rising edge. On the next falling edge, the value of `a` is then assigned to `b`. Since `p0` and `p1` cannot pass their wait statements simultaniously, the IL representation can be written as

```
rise clk -> a := c;
fall clk -> b := a
```

Example 6

In this example, both wait statements are passed simultaniously and the semantics can only be given with respect to a resolution function.

```
signal a,b,c;

p0 : process
 begin
  wait on clk until clk = '1'
   a <= c;
 end process p0;

p1 : process
 begin
  wait on clk until cls = '1'
   a <= b;
end process p0;
```

In this example, signal `a` has two drivers, namely `c` (from process `p0`) and `b` (from process `p1`). In VHDL, a resolution function has to be provided, giving priority to one of all possible drivers. Assuming a resolution function that executes `a <= b` after `a <= c`, the IL semantics is given as

```
@(clk) (a := b)
```

Example 7

Now, delta-delay comes into play:

```
signal a,b,c;

p0 : process (clk)
 begin
  wait on clk until clk = '1'
   a <= c;
 end process p0;

p1 : process (clk)
 begin
  wait on clk until cls = '1'
   b <= a;
 end process p0;
```

Whenever `clk` changes to `1`, `a` is assigned the value of `c` and `b` is assigned the value of `a`. As the asignment strategy of VHDL is based on delta delays, the values of the signals on the right-hand sides remain constant until all assignments have been processed. Only then, the signal values are updated simultaniously.

Thus, the semantics of this program is

```
rise clk -> (a := c);
rise clk -> (b := a)
```

Note that the VHDL program above is *not* equivalent to the Verilog program

```
always @(posedge clk) begin a = c; b = a; end
```

since in Verilog, `a` is updated before processing the second assignment and therefore has semantics

```
rise clk -> (a := c; b := c)
```

However, the Verilog program

```
always @(posedge clk) a = c; b = a; end
```

has a counterpart in VHDL when defining `a` of type *variable* instead of type *signal*:

```
signal b,c;

p0 : process (clk)
   variable a;
  begin
    wait on clk until clk = '1'
    a = c;
    b <= a;
  end process p0;
```

Here, variable `a` is immediately assigned the value of signal `c` and therefore, signal `b` will also become equal to `c`. Thus, the semantics in IL also is

```
  rise clk -> (a := c; b := c)
```

References

[1] ANSI/IEEE Std 1076-1987. *IEEE Standard VHDL Language Reference Manual*. IEEE, New York, USA, March 1988.

[2] D. Borrione, L. Pierre, and A.M. Salem. Formal Verification of VHDL Descriptions in the Prevail Environment. *IEEE Design & Test of Computers*, pages 42–55, June 1992.

[3] R. Boulton, A. Gordon, M.J.C. Gordon, J. Herbert, and J. van Tassel. Experience with embedding hardware description languages in HOL. In *International Conference on Theorem Provers in Circuit Design (TPCD)*, pages 129–156, Nijmegen, June 1992. IFIP TC10/WG 10.2, North-Holland.

[4] R. Boulton, M.J.C. Gordon, J. Herbert, and J. van Tassel. The HOL verification of ELLA designs. In *ACM International Workshop on Formal Methods in VLSI Design*, Miami, 1991.

[5] D. Boussebha, N. Giambiasi, and H. Magnier. Temporal Verification of Behavioral Descriptions in VHDL. In *European Design Automation Conference*, pages 692–697, Hamburg, March 1992. IEEE.

[6] G. Döhmen. Petri Nets as Intermediate Representation between VHDL and Symbolic Transition Systems. In *European Design Automation Conference*, pages 572–577, Grenoble, France, September 1994. IEEE Computer Society, IEEE Computer Society Press.

[7] D. Eisenbiegler, K. Schneider, and R. Kumar. A Functional Approach for Formalizing Regular Hardware Structures. In J.J. Joyce and C.-J.H. Seger, editors, *Higher Order Logic Theorem Proving and its Applications*, volume 780 of *Lecture Notes in Computer Science*, pages 99–113, Vancouver, Canada, August 1993. University of British Columbia, Springer-Verlag, published 1994.

[8] H. Eveking. *Verifikation digitaler Systeme*. Teubner Verlag, 1991.

[9] F.S. Fernandez, P.T. Breuer, and C. Delgado Kloos. Proof Theory and a Validation Condition Generator for VHDL. In *European Design Automation Conference*, pages 512–519, Grenoble, France, September 1994. IEEE Computer Society, IEEE Computer Society Press.

[10] K.G.W. Goossens. Embedding a CHDL in a proof system. LFCS Report Series ECS-LFCS-91-155, University of Edinburgh, May 1991.

[11] M. J. C. Gordon. Various draft papers on the formal semantics of Verilog HDL arising from the VFE project. Available on the web at: `http://www.cl.cam.ac.uk/users/mjcg/Verilog/V/V.html`.

[12] M. J. C. Gordon. Event and cycle semantics of hardware description languages. Unpublished draft paper. Available on the WWW at: `http://www.cl.cam.ac.uk/users/mjcg/Verilog/V/HDLpaper.ps`.

[13] M. J. C. Gordon. Why higher-order logic is a good formalism for specifying and verifying hardware. In G. Milne and P. A. Subrahmanyam, editors, *Formal Aspects of VLSI Design*, pages 153–177. North-Holland, 1986.

[14] M. J. C. Gordon and A. Ghosh. Language independent RTL semantics. Draft of an invited contribution to the 1998 IEEE CS Annual Workshop on VLSI: System Level Design in Orlando Florida. The paper was not presented, due to flights into Orlando being full! Available on the WWW at: `http://www.cl.cam.ac.uk/users/mjcg/OrlandoPaper.ps`.

[15] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: A Theorem-proving Environment for Higher-Order Logic.* Cambridge University Press, 1993.

[16] A. Gupta. Formal Hardware Verification Methods: A Survey. *Journal of Formal Methods in System Design*, 1:151–238, 1992.

[17] Nicolas Halbwachs. LUSTRE is a functional stream-based synchronous declarative language for programming reactive systems: `http://www-verimag.imag.fr//SYNCHRONE/lustre-english.html`.

[18] J. M. J. Herbert. Temporal abstraction of digital designs. In G. J. Milne, editor, *Proceedings of the IFIP WG 10.2 Working Conference*, pages 1–25. North-Holland, Glasgow, July, 1988.

[19] Myra Van Inwegen. Two reports: one on Verilog's expression evaluation and one on Verilog's expression normalization. Available on the web at: `http://www.cl.cam.ac.uk/users/mvi20/handel/verilog.html`.

[20] D.J. Kinniment and A.M. Koelmans. Modelling and verification of timing conditions with the Boyer Moore prover. In *International Conference on Theorem Provers in Circuit Design (TPCD)*, pages 111–127, Nijmegen, June 1992. IFIP TC10/WG 10.2, North-Holland.

[21] B. Levy, I. Filippenko, L. Marcus, and T. Menas. Using the state delta verification system (SDVS) for hardware verification. In *International Conference on Theorem Provers in Circuit Design (TPCD)*, pages 337–360, Nijmegen, June 1992. IFIP TC10/WG 10.2, North-Holland.

[22] T. Margaria. Hierarchical mixed-mode verification of complex FSM's described at the RT level. In *International Conference on Theorem Provers in Circuit Design (TPCD)*, pages 129–156, Nijmegen, June 1992. IFIP TC10/WG 10.2, North-Holland.

[23] K. L. McMillan. The SMV language, June 1998. Available on the WWW at: `http://www-cad.eecs.berkeley.edu/~kenmcmil/psdoc.html`.

[24] Kenneth L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, 1993.

[25] T. F. Melham, editor. *Higher Order Logic and Hardware Verification.* Cambridge Tracts in Theoretical Computer Science 31. Cambridge University Press, 1993.

[26] S. Olcoz and J.M. Colom. A petri net approach for the analysis of VHDL descriptions. In G.J. Milne and L. Pierre, editors, *Conference on Correct Hardware Design and Verification Methods (CHARME)*, volume 683 of *Lecture Notes in Computer Science*, pages 15–26, Arles, France, May 1993. IFIP WG10.2, Springer-Verlag.

[27] L. Pierre. From a HDL description to formal proof systems: Principles and mechanization. In D. Borrione and R. Waxman, editors, *IFIP Conference on Computer Hardware Description Languages and their Applications (CHDL)*, pages 21–41. North-Holland, Marseille, April 1991.

[28] S. Rajgopal, K. Hedlund, and D. Reeves. Integrating hardware verification with CHDL's. In D. Borrione and R. Waxman, editors, *IFIP Conference on Computer Hardware Description Languages and their Applications (CHDL)*, pages 153–163. North-Holland, Marseille, April 1991.

[29] D.M. Russinoff. Specification and verification of gate–level VHDL models of synchronous and asynchronous circuits. In E. Börger, editor, *Specification and Validation Methods*. Oxford University Press, Oxford, 1994.

[30] Synopsys, Inc. *HDL Compiler for Verilog Reference Manual, Version 3.5*, September 1996.

[31] J.P. Van Tassel. A formalisation of the VHDL simulation cycle. In *Higher Order Logic Theorem Proving and its Applications*, pages 213–228. IFIP WG 10.2, September 1992.

[32] Donald E. Thomas and Philip R. Moorby. *The Verilog Hardware Description Language*. Kluwer Academic Publishers, 3rd edition, 1996.

[33] J. van Tassel and D. Hemmendinger. Toward Formal Verification of VHDL Specifications. In *IFIP International Workshop on Applied Formal Methods for Correct VLSI Design*, pages 409–418. IFIP WG 10.2/WG 10.5, North-Holland, 1990, 1989.

[34] X. Wang and E.P. Stabler. Formalization of VHDL synthesis procedures in HOL. In M. Archer, J.J. Joyce, K.N. Levitt, and P.J. Windley, editors, *Higher Order Logic Theorem Proving and its Applications*, pages 106–120, Davis, California, August 1991. IEEE Computer Society, ACM SIGDA, IEEE Computer Society Press.