Clean formal semantics for VHDL[*]

Peter T. Breuer   Luis Sánchez Fernández   Carlos Delgado Kloos

ETSIT, Univ. Politécnica de Madrid, E–28040 Madrid, Spain
<{ptb,lsanchez,cdk}@dit.upm.es>

Abstract

A simple formal semantics for the standard hardware description language VHDL *is set out in* functional *style. The presentation comprises an executable specification for a synchronously clocked* VHDL *simulator.*

1 Introduction

This paper attempts to lay the basis for simpler formal semantics for VHDL than available in the literature. Vhdl is a standardized hardware description language developed in the 1980s with the support of the US DoD, and it has gained wide acceptance. No formal semantics underpins the standard [4], and there has been great interest in developing a formal semantics for the *simulation model* it defines. Timing aspects are formalized in [7], but [9] appears to be the most complete and satisfactory formalization to date. It sets out an *asynchronous* operational semantics for VHDL simulators. In contrast, we set out a *synchronous* semantics. Vhdl is seen as a side-effect on a clocked sequence of global states; one for every unit interval of time.

A good formal semantics is important for many reasons: it serves as a lynchpin for future designs of tools and may reduce the number of idiosyncratic differences between compliant tools from different vendors. It makes the language accessible to modern computing aids such as automated design verifiers and simulators/debuggers, and it may help make the language accessible to educators and students alike.

The model presented here is intended to satisfy these criteria. It is very clean. It is written in a generic functional style [1] and, indeed, the model is concise enough for the specification to be included almost in its entirety. In consequence, the paper itself is executable. It is a valid literate script for the GOFER [5] lazy functional programming environment. Lines beginning with a greater-than sign, >, are processed by the interpreter, and the interpretation of those lines in this paper, plus a few library functions, comprises a simulator for VHDL kernel code as it stands. We believe that our *model-based* design is easier to grasp than extant logical presentations, and that its equational style succeeds in avoiding the excess detail present in other models. Moreover, current work by the authors has successfully employed its logic as a basis for the synthesis and validation of VHDL.

Some features of VHDL have to remain untreated in this short paper; we leave inertial and delta-delayed assignment aside. The treatment of the former is simple, but the latter is not (in this setting). Attributes and resolution functions are also not treated here.

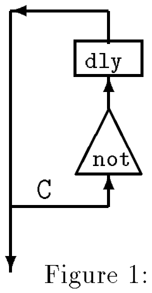
The paper has the following layout. In Section 2 a short description of VHDL and GOFER is given. Section 3 gives the functional presentation of the semantics and its implementation as a VHDL simulator. Section 4 discusses other models of VHDL in the literature.

2 Vhdl and GOFER

Vhdl is a procedural language in the imperative style of Pascal and C. A script is a set of *process* definitions, each of which contains procedural code. The concept is that processes represent hardware modules that execute independently and asynchronously and communicate through *signal wires*. Each process description specifies the wires it is connected to in the header. The VHDL description and hardware diagram of process `a` in Fig. 1 will be alluded to again in this paper. It is a self-oscillator with half-period `dly` outputting 1/0 on signal wire `C`.

The sequential algorithm in the process description

[*]The work reported here has been partially supported by the INTEGRAL project (TIC93–0515–C02–01) sponsored by the Spanish "Comisión Interministerial de Ciencia y Tecnología" and the EuroForm network (2/ERB4050PL921826) sponsored by the Human Capital and Mobility program of the CEC.



```
PROCESS
BEGIN
  C <= TRANSPORT (NOT C) AFTER dly;
  WAIT ON C ;
END
```

Figure 1: Process ‘a’ VHDL code and diagram.

can contain `while` loops, `if` branches, and so on, as well as assignments to local variables. The code inside the process body loops continuously. There are two special atomic statement types in VHDL:

i] *Signal assignment*, an example of which is the first statement in the body of process `a` in Fig. 1. This launches a request for a future assignment to an output signal, prescribing the delay. If the delay is zero, the assignment is called *delta-delayed.*

ii] The `wait` statement, of the general form:

`wait on` *signals* `until` *test* `for` *time*

Signals is the list of signals to which the `wait` statement is sensitive. When a level transition occurs in one such signal the *test* is evaluated. If `true`, the process will resume, else it will remain suspended. *Time* is a timeout value. A `wait` occurs in the Fig. 1 code.

The standard VHDL simulation cycle runs each process of a VHDL description until it becomes blocked in a `wait` statement. Zero logical time has elapsed to this point. Logical time is then advanced to the next pending assignment time, and the assignment executed. The conditions of each `wait` statement are then checked, and if any are satisfied the appropriate process executions are resumed (in zero logical time) until they all become blocked again. Then logical time is advanced again, and so on.

We have chosen the GOFER [5] programming environment in which to define the semantics of VHDL. GOFER is a modern pure functional language (i.e. a language based on expressions rather than statements with program variables) which is syntactically and semantically close to the HASKELL standard [3]). See [1] for a good introduction to declarative languages of this type. GOFER evaluates the elements of potentially infinite data structures only on demand (it is *lazy*); it has higher-order functions as ordinary datavalues; polymorphic typing, and many other features that suit it to modelling an embedded language. Note that we do not depend particularly on the semantics of GOFER, but on the generic λ-calculus of functions on which all functional languages are based. GOFER expressions are syntactic sugar for λ-calculus terms.

Fig. 2 shows some small examples of GOFER programs. Type declarations (with an infix `::`) are not mandatory, as the type will be inferred by the system, but they are a useful notational aid. The type `a->b` is the type of functions with domain `a` and codomain `b`. The type `[a]` is the type of lists with elements of type `a`. Tuple types are written `(a,b)`, `(a,b,c)`, and so on. There are no singleton tuples.

3 Functional Semantics

The semantics of VHDL statements proposed here is a side-effect on a sequence of partially defined future states called a *driver set* (Δ), yielding as result the sequence of states that existed during the time that the statement blocked for. This is called an *episode* (ϵ). Both signal assignment and `wait` statements are treated explicitly, and these two are the only atomic statement types in this paper. For simplicity (as in [9]), delta-delayed signal assignments are *not* treated.

There is no state variable information, and no set of events (γ in [9]). A further simplification is that state variables are treated like signals – the gap in the functionality of delayed assignment left by the omission of delta delays turns out to precisely accomodate variable assignment. In the sequel, we assume that the nanosecond is the standard unit, and write 1 ns for a delay of one unit time.

We rely on the fact that more complex VHDL descriptions can be reduced to essentially this simple subset, as reported in [8]. The following statement and process combinators will be needed for a corresponding VHDL *pseudo-code*: `_%%_` (sequence), `_%|_` (parallel), `%=` (assign) and `wait`, `_%?_%:_` (conditional branch) and `process`. The symbols beginning with a percent are infix.

The `process` combinator forms a process from a (possibly compound) statement. Processes loop continuously. Concurrent execution (`%|`) is the only process operator. (`%%`) is the sequential operator between statements, assignment (`%=`) and `wait` are statement constructors, `_%?_%:_` is the conditional branching combinator on statements. A single process will typically have the concrete VHDL pseudo-code:

```
process [inputs] [outputs]
begin(
  statement1  %%
```

```
 Fibonacci numbers:  fib :: Int -> Int              -- This is a comment.

> fib 0     = 0                                     -- Using pattern matching:
> fib 1     = 1                                     -- base cases ...
> fib (n+2) = fib n + fib (n+1)                     -- recursive case.

 Prime numbers:       primes :: [Int]

> primes       = sieve [ 2 .. ]                     -- Using comprehensions. Dots generate sequences
> sieve (p:xs) = [ x | x<-sieve xs, rem x p /= 0 ] -- and rem is remainder after integer devision.

 Polymorphism:        iter :: (a->a) -> a -> [a]    -- Lower case denotes type variables.

> iter f x = x:iter f (f x)                         -- Informally [ x, f x, f(f x), .. ]
```

Figure 2: Examples of GOFER programs.

```
Statement ::= Id %= Expression after Delay            Process   ::= process Id* Id* begin Statement end
            | wait on Id*                                         | Process %| Process
            | Statement %% Statement
            | Expr %? Statement %: Statement
```

Figure 3: The BNF for the concrete syntax of VHDL pseudo-code.

```
    ...          %%
    statementN
  )end
```

and the complete Backus-Naur Form for the concrete syntax of the pseudo-code language is given in Fig. 3.

Nearly the same set appears in [9], where it is called femto-VHDL. The `wait` statement is an addition here, as is the appearance of named inputs and outputs within the `process` constructor. Other standard combinators, such as `while`, would be treated in the same manner as conditional branching will be treated here.

Statements are side effects on a driver set Δ, yielding an episode ϵ.

```
> type StatementSemantics =
>       DriverSet -> (Episode,DriverSet)
```

The episode is a finite sequence of states (for terminating code). It records those values assigned between the time that control passed to the program and the time that it left it, as well as those values undisturbed. The time sequence is shown in Fig. 4.

```
> type Episode   = [State]
>       --  a sequence of states
```

The driver set is a pair of infinite sequences of states in time order, divided by the pairing into information about future inputs and future outputs.

```
> type DriverSet = (Inputs,Outputs)
>       --  pair of inputs and outputs
> type Inputs    = [State]
>       --  sequence of input components of state
> type Outputs   = [State]
>       --  sequence of output components
```

A *state* binds identifiers to values:

```
> type State    = [(Id,Val)]
>       --  set of identifier/value pairs
```

To be definite, we use strings as identifiers for the signal wires, and integers as the values assigned to them. Booleans will be represented by 1 and 0.

```
> type Id       = String
> type Val      = Int
```

For all the driver sets we consider, the inputs and outputs should be disjoint states at corresponding time positions. This reflects the idea that information should not be forced into a single channel from two different sources.

If a signal is both notionally an input and an output (i.e., it is part of a feedback loop), then it is included in the outputs from that point in time at which the feedback first becomes defined, and in the inputs until then. The sequence starts with information about the notional current state (now+0 ns), then continues with information about the states at now+1ns, now+2ns, and so on, as shown in Fig. 4.

The state comprised by position 1 in the driver set components is the first future state. We are morally permitted to recalculate the forthcoming state. The position 0 state, on the other hand, is the current

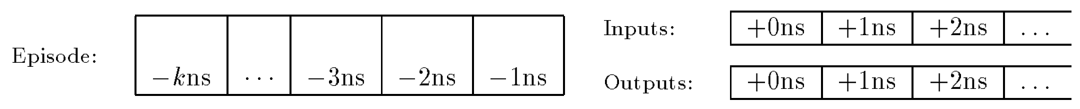


Figure 4: The time sequence of states within a driver set and an episode.

State	::=	*State*	`+@`	*State*	– – overwrite from left to right
	|	*State*	`@+`	*State*	– – overwrite from right to left
	|	*Id**	`@=`	*Val**	– – create by assignment
	|	*Id**	`@:`	*State*	– – restrict to given domain
	|	*State*	`+@+`	*State*	– – sum of disjoint states

Figure 5: The BNF for the concrete syntax of *State* constructions.

state. We may not change it, except by the addition of derived information, or of information on which no parallel process depends (i.e. local variable updates are permitted, but not signals).

A process is a map from an initializing input driver set to a final output driver set which records the lifetime process outputs, plus the names of the input and output signals.

```
> type ProcessSemantics =
>      (Domain,Codomain,Inputs->Outputs)
> type Domain   = [Id]
> type Codomain = [Id]
```

The names are needed when combining processes.

3.1 Operations of state

The BNF for states is shown in Fig. 5. The semantics for the state operators appears in Appendix A.

We can *overwrite* components of the state `s` with those from a partial state `s'` using the notation `s'+@s`. Or we can write it the other way round; `s@+s'`. Both operations may properly extend the domain of the state. We can also *combine disjoint* partial states `s1` and `s2` using the operation `s1+@+s2`. Disjoint means that the domains of the two states have no identifier in common. Part states can be *created* using the assignment operator `@=`. We can *look up* current assignments in a state with `@!`, and *restrict* a state to a specified domain with `@:`. If `x` is an identifier and `y` is a value, then `[x]@=[y]` denotes the state with the singleton domain `x` which binds `y` to `x`. `[x1,x2]@=[y1,y2]` binds `y1` to `x1` and `y2` to `x2`, and so on. Given the state `s`, `s@!x` looks up the value bound to `x` in `s`, and so `([x]@=[y])@!x` is `y`. Given the state `s`, `[x]@:s` is the substate with singleton domain `[x]`, `[x1,x2]@:s` is the substate with domain `[x1,x2]`, and so on (provided that `x`, `x1`, `x2` were all in the domain of `s`).

```
>--  structure   index   field type
>--           ↘    ↓   ↙
> class Record a b c where     -- The Record type class
>    (!) :: a -> b -> c        -- has type triples with
>    (!=):: b -> c -> a -> a   -- two operators defined
>                              -- per triple: lookup (!)
>                              -- & assignment (!=).

> instance Record [a] Int a where
>    (!) = (!!)                -- List lookup & writing
>    (x!=y) l = l1++y:tail l2 -- is indexed by offset.
>               where (l1,l2) = splitAt x l

> instance Record State Id Val where
>    (!) = (@!)                -- State lookup & write
>    (x!=y)s = s@+([x]@=[y])  -- is indexed by Id.
```

Figure 6: A uniform interface to states and lists.

Lookup can be extended to expressions in the obvious way. We write `s@@e` when `e` is an expression, and `s@!i` when `i` is an identifier. For definiteness, here is the actual definition of the data type of expressions (i.e., not the BNF) in GOFER. `Binop` is the type of integer-valued binary operations on integers, and `Monop` is the type of 1-ary operations.

```
> data Expression
>       = V Id                   -- Variable
>       | K Int                  -- Constant
>       | M Monop Expression     -- 1- and 2-ary operations
>       | B Binop Expression Expression
```

The letters V, K, M, B are the labels on the nodes of the syntax tree for each distinct kind of expression. `2+2` is represented as follows: `B (+) (K 2) (K 2)`.

A convenient uniform interface to both states and lists can be produced by defining a common *type class* called *Record* of which both states and lists are instances, and which admits both lookup (`!`) and assignment (`!=`). This is shown in Fig. 6. The interface can be extended further to allow the use of lists of identifiers or integers for indexing. The symbols (`!`, `!=`) will be used instead of the list lookup (`!!`) and state lookup (`@!`) and state assignment (`@=`) or overwrite (`@+`) whenever possible.

3.2 Statement semantics

Statements compose in sequence with the semantics of a side-effecting concatenation of the episodes that they produce, with the leftmost done first.

```
> (bs1 %% bs2) dlt = (eps++eps', dlt'')
>      where (eps ,dlt' ) = bs1 dlt
>            (eps',dlt'') = bs2 dlt'
```

What we are really interested in in VHDL is signal assignment and **`wait`** statements. Assignment is non-blocking, so the episode it delivers is empty. It overwrites at the specified future state, now+`t`ns, in the given driver set. It also pre-empts later signals to `x`, and that can be expressed here by making it write at *all* future times too. This depends on lazy semantics for its executability.

```
> (x %= ex `After` t) (is,os) =
>        ([], (is,os1++map (x!=ex') os2))
>        where (os1,os2) = splitAt t os
>              ex'= s0!ex -- value of ex in s0
>              s0 = (is!0) +@+ (os!0)
```

The current state is used for the calculation.

A **`wait`** statement does block, so it emits a nonempty episode. But it does not block long if the current state of the variables `xs` differs from the immediately future state, in which case it emits the current state. The state in which the transition has just happened is then current. Recall that we have restricted the syntax to prevent delta-delayed assignments to signals, which might otherwise undo the transition.

```
> wait on xs (is,os) = (s0:eps, dlt)
>   where
>   (eps,dlt) = if y0==y1 then
>               wait on xs(tail is,tail os)
>               else ([],(tail is,tail os))
>   s0 = (is!0) +@+ (os!0)
>   s1 = (is!1) +@+ (os!1)
>   y0 = s0!xs
>   y1 = s1!xs
```

There should be no information conflicts between inputs and outputs, because they are disjoint.

Now conditionals, in the straightforward manner. The tested expression is looked up in the current state `s0`. If it is zero, `bs0` is the statement chosen for execution. If equal to 1, then `bs1`.

```
> (e %? bs1 `Or` bs0) (is,os) =
>   (if (v/=0) then bs1 else bs0) (is,os)
>   where s0= (is!0) +@+ (os!0)
>         v = s0!e
```

Other control constructs can be treated in the same way. Indeed, since infinite pseudo-code structures have not been forbidden, while loops are already expressible as infinitely nested conditionals.

3.3 VHDL process semantics

The **`process`** constructor makes a process out of a statement. What is executed is essentially `bs%%bs%%bs%%...`, but only the resulting episode is of interest. The final output driver set is never attained, because the process is continuous.

```
> process i o Begin bs End = (i,o,ps)
>    where ps = filter2.fst.loop bs.filter1
>          loop bs     = bs %% loop bs
>          filter2     = map (o@:)
>          filter1 is  = (map (i@:) is,os)
>          os          = emptyStateSeq
```

The sequence of empty states, **`emptyStateSeq`**, serves as the initial (provisional) output driver set `os`.

Concurrency should be simple, but we have had to experiment to get it exactly right. The simplest obvious possibility, **`naivepar`**, takes two processes with disjoint outputs and runs them simultaneously, merging the output episodes back together again pointwise. The diagram is shown in Fig. 7.

But this formulation is inadequate when the two processes depend on each others outputs, as in, for example, a bistable oscillator. We would wish to feed the output back in so that the two processes can see each others signals. The simplest solution is **`optimisticpar`** (see Fig. 7). Different processes never write to the same ports, so the **`+@+`** operator can be used to combine the outputs.

Although **`optimisticpar`** expresses the idea that the parallel combination **`ps`** of two processes **`ps1`** and **`ps2`** must interact with feedback, experiment has shown that the feedback is applied with insufficient discrimination. If process **`ps1`**, say, has internal feedback, then the new loop supplies that feedback one more time. We correct this in **`par2`**, introducing two screened feedback loops. The outputs of the second process are fed back to the inputs of the first process, and vice versa. See Fig. 7.

```
> par2 (i1,o1,ps1) (i2,o2,ps2) = (i,o,ps)
>   where
>   (i,o) = (i1 ∪ i2, o1 ∪ o2)
>   ps is = os
>     where
>     os  = zipWith (+@+) os1 os2
>     os1 = ps1 is1
>     os2 = ps2 is2
>     is1 = zipWith (+@) is os2'
>     is2 = zipWith (+@) is os1'
>     os1'= [(o1∩i2)@:(os1!!n)|n<-[0..]]
```

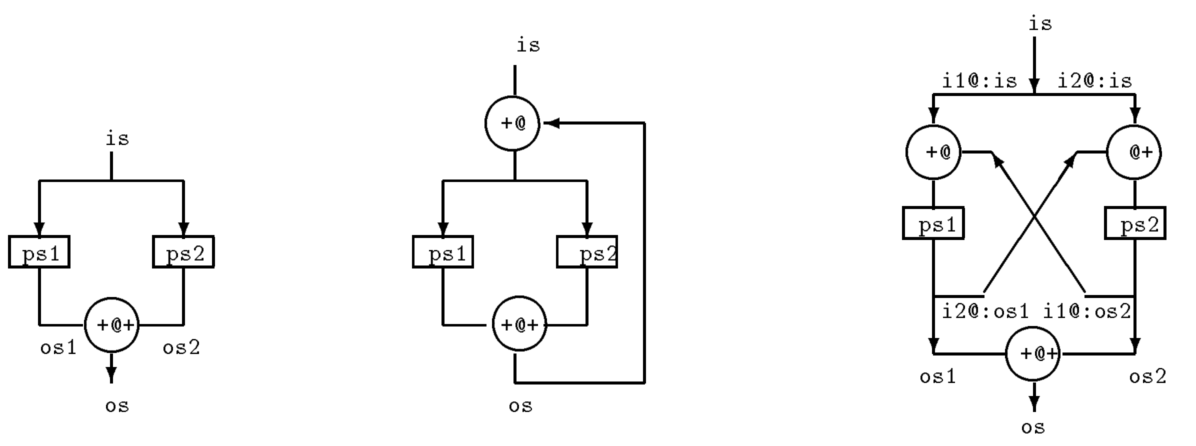


Figure 7: Wiring diagrams for `naivepar` (left), `optimisticpar` (centre), par2 (right).

```
> a = process ["C"] ["C"] begin(a1%%a2)end -- PROCESS
> a1= "C" %= M inot (V "C") `after` dly    -- BEGIN
>     where inot x = 1-x                   --   C<=TRANSPORT 1-C AFTER dly ;
> a2= wait on ["C"]                        --   WAIT ON C ;
>                                          -- END

> b = process ["C"] ["D"] begin(b1%%b2)end -- PROCESS
> b1= "D" %= V "C" `after` 0               -- BEGIN
>                                          --   D<=TRANSPORT C AFTER 0 ;
> b2= wait on ["C"]                        --   WAIT ON C ;
>                                          -- END
```

Figure 8: Pseudo code representations of VHDL processes ‘a’ and ‘b’.

```
>     os2'= [(o2∩i1)@:(os2!!n)|n<-[0..]]
```

We have had to force the generic fixpoint calculation up a little in order to generate streams. But this is the operator to use.

```
> (%|) = par2
```

Now to duplicate the example in [9]. This is a uniprocess oscillator, `a` with half-period `dly` (Ins=Outs=C), and a follower, `b` (Ins=C, Outs=D), and their parallel composition, `c` (Ins=C, Outs=C,D).

```
> dly = 1
```

Process `a` has the VHDL code given in Figure 1, and it will be written as a continuous pseudo-code loop with the representation shown in Figure 8.

The code makes C oscillate between 1 and 0 with half-period 1. The output is shown in Table 1. We start with input driver set `f` in which only the current state is asserted.

```
> s0= ["C","D"]@=[0,0]
> f = s0:emptyStateSeq
> emptyStateSeq = ([]@=[]):emptyStateSeq
```

Process `b` is represented by the pseudo-code loop also shown in Figure 8. Process `c` is both together:

```
> c = a %| b
```

The `a` and `b` components have disjoint outputs.

The trace of the follower on its own is shown in Table 1. It never sees C change so the output is constant. It is trapped in the implicit `wait` at the end of the process loop. The two running together are also shown in Table 1. D follows C in the same ns interval.

4 Other models of VHDL

In [9], the operational semantics of VHDL is given as a side-effect on two components: the state variables σ, and a set τ of scheduled updates to future signals. The stand-alone `wait` statement is not treated. In contrast, the `wait` statement is treated in the present paper and the semantics is declarative, not operational, although it is executable. The environment in [9] explicitly contains the set γ of currently occurring signal transitions, and this information is derived here.

process `a`	process `b`

`? trace a f`
`1) C=0`
`2) C=1`
`3) C=0`
`4) C=1`
`...`
	`? trace b f`
	`1) D=0`
	`2) D=0`
	`3) D=0`
	`4) D=0`
	`...`

inputs `f`	`a`, `b` in parallel

`? showInputs f`
`1) C=0, D=0`
`2)`
`3)`
`4)`
`...`
	`? trace c f`
	`1) D=0, C=0`
	`2) D=1, C=1`
	`3) D=0, C=0`
	`4) D=1, C=1`
	`...`

Table 1: Processes `a` and `b` singly and in parallel.

Two interpretations in Petri nets are given by [2]. The first resynchronizes the Petri nets representing processes at every instant of logical time, which matches the spirit of the model given here. The second semantics is based on communication of a driver set between otherwise independently operating Petri nets, and in that way (only) resembles the model given here. It must be said that both semantics generate large Petri net equivalents from small VHDL definitions. Another Petri net interpretation is reported in [6], but this is a direct translation of the simulation semantics in the VHDL standard and it contains an operating system kernel exercising centralized control. It is not compositional (except modulo the kernel semantics!).

5 Summary

A synchronous declarative semantics for VHDL has been presented. It is functionally executable as a VHDL pseudo-code simulator as it stands.

The semantics is based on a model of VHDL statements as side-effecting on a schedule of future global states, delivering as result a sequence of past states.

References

[1] R. Bird and P. Wadler. *Introduction to functional programming.* Prentice Hall International, 1989.

[2] W. Damm et al. A formal semantics for VHDL based on interpreted petri nets. TR, Univ. Oldenburg, 1992.

[3] P. Hudak, P. Wadler, et al. Report on the programming language Haskell, a non-strict purely functional language. TR, Yale/Glasgow Univ., Aug. 1991.

[4] IEEE, *Standard VHDL Language Reference Manual*, 1988. IEEE Std 1076-1987.

[5] M. P. Jones. Introduction to Gofer. TR, Dept. Comp. Sci., Yale, USA, September 1991. (Part of the Gofer distribution, available by anonymous ftp from nebula.cs.yale.edu:pub/haskell/gofer).

[6] S. Olcoz and J. Colom. Petri net based analysis of VHDL descriptions. In *2nd Intl. Conf. EuroVHDL 91*, Swed., Sept. 1991.

[7] A. Salem and D. Borrione. Formal semantics of VHDL timing constructs. In *VHDL for simulation, synthesis & formal proofs of hardware*, ed. J. Mermet, Kluwer, 1992.

[8] L. Sánchez and C. Delgado Kloos. Functional description of VHDL. In *Segundo Congreso de Programacion Declarativa PRODE 93*, Spain, Sept. 1993.

[9] J. van Tassel. A formalization of the VHDL simulation cycle. TR 249, Univ. Cambridge, Comp. Lab., 1992.

A State operations

```
> (+@)  :: State->State->State
> s' +@ s = s' +@+ (s @- s')      -- overwrite from left

> (@+)  :: State->State->State
> s @+ s' = s' +@ s               -- overwrite from right
```

Here `@-` removes that part of the domain of the left hand side which also is part of the domain of the right:

```
> (@-)  :: State->State->State -- anti-restrict from right
> s @- s' = (dom s \\ dom s') @: s

> (-@)  :: State->State->State -- anti-restrict from left
> s -@ s' = s' @- s

> (+@+) :: State->State->State -- disjoint sum
> (+@+) = (++)

> (@=)  :: [Id] ->[Val]->State -- create by assignment
> xs @= ys= zip xs ys

> (@!)  :: State->Id   ->Val   -- lookup Identifier
> s @! x  = head [y| (x',y)<-s, x==x' ]

> (@:)  :: [Id] ->State->State -- restrict
> xs @: s = [(x,s@!x)|x<- xs ∩ dom s]

> dom   :: State->[Id]         -- identifiers in domain
> dom s   = [x| (x,y)<-s ]
```

The definition of restriction (`@:`) is very sensitive.

The following auxiliary functions on lists are required.

```
> (\\) :: Eq a => [a] -> [a] -> [a]
> (x:a)\\b = if elem x b then a\\b else x:(a\\b)
> []   \\b = []                   -- difference

> a ∩ b = a \\ (a \\ b)          -- intersection
```