A Refinement Calculus for VHDL

Peter T. Breuer Carlos Delgado Kloos
Natividad Martínez Madrid Andrés Marín Luis Sánchez
Departamento de Ingeniería de Sistemas Telemáticos, ETSI Telecomunicación
Universidad Politécnica de Madrid, Ciudad Universitaria, E–28040 Madrid, Spain
<ptb,cdk,nmadrid,amarin,lsanchez@dit.upm.es>

Abstract

A refinement calculus for the specification of real-time systems and their refinement to a VHDL behavioural description is set out here. The specification format is a logical triple with the look of a Z or VDM schema. Choices from a short menu of refinement operations gradually convert an initial specification to VHDL code through a series of mixed mode intermediates. The calculus is complete in the sense that if there is a code of the VHDL subset considered here (unit-delay waits and signal assignments but no delta delays) satisfying the specification, then it can be obtained by applying some sequence of the refinement operations. The result is "correct by construction".

1. Introduction

This article describes a refinement method for VHDL. It is intended to produce behavioural code that is "correct by construction", departing from a non-behavioural specification. It is a formal method based on a formal language and formal rules. It consists of a specification language $\mathcal{L}$ supporting a relation $\sqsubseteq$ ('refines to') plus a set of partial operations $\mathcal{O} \subseteq \mathcal{L} \nrightarrow \mathcal{L}$. The triple $(\mathcal{L}, \sqsubseteq, \mathcal{O})$ is a *refinement calculus.* This means that the result of applying an operation $op \in \mathcal{O}$ to a specification a is a refined specification $b = op(a)$:

$$op \in \mathcal{O} \wedge a \in \mathrm{dom}\, op \rightarrow a \sqsubseteq op(a)$$

The refinement relation is transitive, so a sequence of such operations leads to a result that is still a refinement of the initial specification:

$$a \sqsubseteq op_1(a) \sqsubseteq op_2(op_1(a)) \sqsubseteq \ldots = b \rightarrow a \sqsubseteq b$$

The usefulness of this system derives from the fact that $\mathcal{L}$ contains specifications which range from the purely logical through mixtures of predicates and code to pure VHDL code. So specifications can be driven from the abstract to the concrete by the sequence of operations.

The purely logical specifications consist of triples

$$[Pre \mid Dur \mid Post]$$

in which Pre is a pre-condition for the specification to become active, Dur is a condition that is to hold while the specification is active, and $Post$ is a post-condition that must hold when it ceases to be active.

As an example, the following specifies a square wave oscillator with half-period one on signal `Q` (with value Q). T is the distinguished variable of time:

$$[{}^{\backprime}T = 0 \mid Q = \lfloor T \,\mathbf{mod}\, 2 \rfloor \mid false]$$

The backwards prime on the first T (and forward primes on variables in the last compartment of the specification, if any) is a technicality: it allows variable names for the pre-, post- and during- states to be distinguished, as in Z [11] or VDM specifications. It can be disregarded for the puspose of this informal discussion. The initial $T = 0$ makes the oscillator active from system startup. The final *false* means that the oscillator can never halt. The middle condition asserts the value of the output signal in terms of the current time whilst the oscillator is running.

The "pure VHDL code" to which this specification may be refined consists of a variant subset of behavioural VHDL in which both wait statements and signal assignments are allowed, but not zero waits or delta-delayed assignments. Note that the process header names the *output* and *input* signals explicitly. The full syntax is shown in Figure 1. An appropriate code for the oscillator is:

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

EURO-DAC '96 with EURO-VHDL '96
0-89791-848-7/96 $4.00 © 1996 IEEE

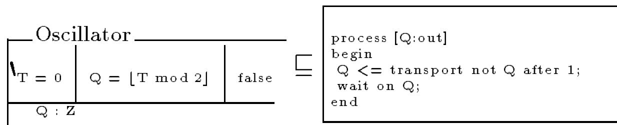
```
ℒ              ::=  Process [ || Process || ... ]
Process        ::=  process [ [ Channels ] ] begin VHDL end
                |   Specification
VHDL           ::=  Statement [ ; Statement ; ... ]
Statement      ::=  Channel <= transport Expression after Delay
                |   wait on Channels
                |   if Expression then VHDL else VHDL
                |   null
                |   Specification
Expression     ::=  Channel
                |   Value
Specification  ::=  [ \Predicate | Predicate | Predicate' [ where [ Decls • ] Predicate ] ]
```

Figure 1. The syntax of $\mathcal{L}$, the language of mixed specification and VHDL code hardware descriptions. See text for additional semantic restrictions.

```
process [Q:out]
begin
   Q <= transport not Q after 1;
   wait on Q;
end
```

It should be emphasized that the process header in this variant subset does *not* denote a standard VHDL sensitivity list. The default time unit is 1 ns.

The refinement relationship $a \sqsubseteq b$ expresses the idea that the behaviour of the code b *satisfies* the specification a. These notions are formalized using a semantic model of unit-delay VHDL set out in [2] and further discussion of this aspect will be found in the following paragraphs. In the foregoing example of the oscillator, the final statement of refinement is:



using the "vertical" ("schema") format for the pure logical specification.

The refinement calculus is *relatively complete*, in that, for example, if there is a unit-delay VHDL coding of a design that can be proved to satisfy a requirements specification, then it can be obtained constructively via these refinement operations. The relativeness is with respect to the power of the underlying logic. We have to allow all facts of the logic to be available as axioms, without further proof.

Solid theoretical support for the practice of real-time programming and hardware design has a much shorter history than that of its cousins supporting the development of software programs for non-time critical applications. The following are the major approaches.

With respect to formal methods for real-time programming [6], an obvious "upgrade path" from formal methods for un-timed programs is to treat time as a read-only shared variable, and then proceed as for un-timed programs [10].

Specifications using *temporal propositional logic* (that is, a modal logic of the time variable in place of unconstrained first order quantifications) have recorded significant successes using *model checking* techniques [4] over the last ten years, primarily due to the efficiency of the algorithms. For finished finite state machines such as hardware microprocessors the science is solid (although the complexity is daunting), but a strategy for getting to a machine description from a requirements specification is not yet as clear as the strategy for verifying a given machine. Nevertheless, hardware compilation and verification may be approaching a mature stage through this science [1].

Statecharts (timed state transition diagrams) [5] typify another promising school. Time constraints on transitions are specified as intervals in which the transition must occur.

Timing diagrams are a real alternative to the style of specification presented here. Recent efforts have resulted in their proper formalization [9] and it may be possible to tie the representation to the formal specifications used in this article. The significant difference lies with the underlying logic, which is of the "strong" kind there and is necessarily of the "weak" kind here.

In our logic only safety properties of processes may be proved, not liveness properties. The philosophy here is that synthesising hardware from a VHDL code is problematic. If the code is synthesizable, then the processes in the code are automatically live, because hardware cannot evaporate. If the code is not synthesizable, then it is not of interest. Deciding whether the code is synthesizable is a separate problem.

The idea of a formal refinement calculus is not new

in the context of formal methods for software system development (e.g. [8]), but the technique is new in the hardware field. The approach here is related to attempts to phrase the semantics of software programs with interrupts [3] and a recent description of a refinement calculus for programs with multiple exits in their control flow graph [7].

The layout of this article is as follows: Section 2 and 3 introduce the specification and refinement technique through worked examples.

2. Specification

In this section, we consider the derivation of the oscillator code in the Introduction. As discussed there, the appropriate pure specification of the oscillator behaviour is encoded in the following schema:

Oscillator

${}^{\backprime}T = 0$	$Q = \lfloor T \,\mathbf{mod}\, 2 \rfloor$	*false*	(s_0)
$Q : \mathbb{Z}$			

Recall that the backwards prime on the T in the first "compartment" indicates that it is a value of a variable on entry to the process. That the process a implements this specification is written:

$$Oscillator \sqsubseteq \mathtt{a}$$

All constructions of code and specifications preserve the refinement order. That is, for codes or specifications a, b, c, and for any constructor f, such as `while true do` ...:

$$a \sqsubseteq b \ \rightarrow\ f[a] \sqsubseteq f[b]$$

The refinement consists of a series of steps:

$$Oscillator = s_0 \sqsubseteq s_1 \sqsubseteq s_2 \sqsubseteq s_3 = \mathtt{a}$$

in which the stepping stones s_1, s_2, s_3 are hybrids, each consisting partly of a specification and partly of code. Rather than writing the refinement operation in question as the application of a guarded function, we justify each operation below via a formal *refinement law.*

2.1. Initialization and iteration

The process begins at time zero and the output signal Q will be reset to zero at that time, as a convenient initialization before anything else happens. The reset takes no time, and it is not possible to interrupt it, but it does terminate. When it terminates, the process will have set the output line to zero and will also have scheduled the outputs to be zero forever, i.e.:

$$\underline{\textbf{always}}\ Q = 0$$

We suppose that the oscillator is to be implemented as a single process. This is essentially a forever loop preceded by the appropriate initialization. To specify the process interior, a *loop invariant*, I, will be needed.

The invariant holds at the start of each process cycle, at the end of each cycle, and during each cycle. Moreover, it must be set up on entry to the loop:

$$T = 0 \wedge \underline{\textbf{always}}\ Q = 0 \rightarrow I$$

and the invariant must be sufficiently strong to maintain the during-condition of the oscillator specification while the process body is executing:

$$I \rightarrow Q = \lfloor T \,\mathbf{mod}\, 2 \rfloor$$

Let us suppose that the n'th cycle starts at time t_n, for some given sequence of times $0 = t_0 < t_1 < t_2 < \ldots$. Then we are looking for a process body with the following specification:

Oscillator body

${}^{\backprime}I \wedge {}^{\backprime}T = t_n$	I	$I' \wedge T' = t_{n+1}$

The most sensible design is to let each iteration take one unit of time:

$$t_n = n$$

We can set the invariant I to assert that the output Q should be planned to take the proper value *for the remainder of the current unit time interval.* This is a very minimal scheduling requirement. Here, $p\ \underline{\textbf{for}}\ \tau$ is the temporal logic construction that asserts that a condition p is currently holding and is planned to endure for a time τ:

$$I \textit{ is } Q = \lfloor T \,\mathbf{mod}\, 2 \rfloor\ \underline{\textbf{for}}\ (\lceil T \rceil - T)$$

The oscillator can then have the form:

```
process [Q : out]
begin
  Oscillator body                                (s1)
end
```

All that has to be checked is that this choice of invariant does indeed force the oscillator invariant $Q =$

$\lfloor T \,\mathbf{mod}\, 2 \rfloor$ (it does, because $p\,\underline{\mathbf{for}}\,\tau$ implies p), and that it is forced by the oscillator process proper initial condition $T = 0$ *and* $\underline{\mathbf{always}}\, Q = 0$ (it is, because $\underline{\mathbf{always}}\, Q = 0$ implies $Q = 0$ in particular). The following is the general rule:

> **Law 1** *(Process) Let $0 = t_0 < t_1 < \ldots$ be a time sequence increasing strictly to infinity, and let I be an invariant with*
>
> $$T = 0 \wedge \underline{\mathbf{always}}\, Q = 0 \rightarrow I$$
>
> *Then*
>
> $$[\grave{} T = 0 \mid I \mid \mathit{false}] \sqsubseteq \begin{array}{l}\texttt{process}\,[Q]\\ \texttt{begin}\\ \;[`T = t_n \wedge `I \mid I \mid T = t_n \wedge I']\\ \texttt{end}\end{array}$$

This is an instance of a (more general) law concerning while loops, but (for simplicity) while loops within a process body are not allowed by the syntax here. There is no theoretical objection to their inclusion.

2.2. Sequence

We can imagine that the process body does its work in two parts. First it modifies the scheduled output values and then it shifts the current time point to allow the change to take place:

Oscillator body is Schedule $;$ *Wait*

This is the refinement rule:

> **Law 2** *(Sequence)*
>
> $[\grave{}\mathit{pre} \mid \mathit{dur} \mid \mathit{post}']$
> $\sqsubseteq [\grave{}\mathit{pre} \mid \mathit{dur} \mid \mathit{mid}'] \,;\, [\grave{}\mathit{mid} \mid \mathit{dur} \mid \mathit{post}']$

And now the oscillator code has the form:

```
process [Q : out]
begin
  Schedule;                                   (s2)
  Wait;
end
```

The scheduling part can be designed to be non-interruptible and to schedule a change in one unit of time's time. Let q be a logical constant that captures the initial value of the output Q. Suppose that *Schedule* acts at a time when Q is already scheduled to keep this value for at least one unit of time ($Q = q\,\underline{\mathbf{for}}\,1$). Then it can schedule an inversion after that unit of time has passed (using the temporal logic construction $p\,\underline{\mathbf{then}}\,q$ to express "q holds immediately after p passes", $Q = q\,\underline{\mathbf{for}}\,1\ \underline{\mathbf{then}}\ \underline{\mathbf{always}}\, Q = \neg\, q$):

Schedule

$`Q = q\,\underline{\mathbf{for}}\,1$	*false*	$Q' = q\,\underline{\mathbf{for}}\,1\ \underline{\mathbf{then}}$ $\underline{\mathbf{always}}\, Q' = \neg\, q$
$\grave{}T = T = T'$		

The during-condition being *false* means that the operation is not interruptible. We force the termination time to be the same as the start time with $\grave{}T = T' = n$ (for the n'th loop iteration).

The *Wait* part now begins (at time $T = n$) when a change has already been scheduled in one unit of time. It waits for the change to occur:

Wait

$\grave{}Q = q\,\underline{\mathbf{for}}\,1\,\underline{\mathbf{then}}$ $\underline{\mathbf{always}}\,\grave{}Q = \neg\, q$	$Q = q\,\underline{\mathbf{for}}$ $(\lceil T \rceil - T)$	$\underline{\mathbf{always}}$ $Q' = \neg\, q$
$`T \leq T \leq \grave{}T + 1 = T'$		

2.3. Atomic statements

In VHDL the above specifications are implementable directly as:

```
Q ⇐ transport ¬Q after 1;
wait on Q
```

The transport assignment implements the schedule part. The general law is as follows:

> **Law 3** *(Transport) A transport assignment of expression X to Q after a delay τ satisfies a specification if the pre-condition implies the post-condition when the current value x of X replaces the appearance of any scheduled value of Q from τ onwards. That is, provided that*
>
> $$\mathit{pre} \rightarrow \mathit{post}[\mathit{if}\ \ T \geq t + \tau\ \mathit{then}\ x\ \mathit{else}\ Q)/Q]$$
>
> *then*
>
> $$[\grave{}\mathit{pre} \mid \mathit{false} \mid \mathit{post}'] \sqsubseteq Q \Leftarrow \texttt{transport}\ X\ \texttt{after}\ \tau$$

In this instance, the substitution in the post-condition yields exactly:

$$Q{=}q\ \underline{\textbf{for}}\ 1\ \ \underline{\textbf{then}}\,\underline{\textbf{always}}\,\neg\ q{=}\neg\ q$$

which is the same as the pre-condition $Q{=}q\ \underline{\textbf{for}}\ 1$, so

$$Schedule \sqsubseteq \texttt{Q} \Leftarrow \texttt{transport}\ \neg\,\texttt{Q after 1}$$

The wait statement implements the remaining subspecification. The rule is:

> **Law 4** *(Wait) If pre forces the first change in Q to occur at a time when post holds:*
>
> $$pre \rightarrow Q{=}q\ \underline{\textbf{until}}\,post \wedge dur\ \ \underline{\textbf{until}}\ \ Q{\neq}q$$
>
> *then the specification can be implemented by a* `wait on Q` *statement:*
>
> $$[{}^{\backprime}pre \mid dur \mid post'] \sqsubseteq \texttt{wait on Q}$$

Here the pre-condition pre is

$$Q = q\ \underline{\textbf{for}}\ 1\ \underline{\textbf{until}}\ \ \underline{\textbf{always}}\ Q{=}\neg\ q$$

and this implies both $Q{=}q\ \underline{\textbf{until}}\ \ \underline{\textbf{always}}\ Q{=}\neg\ q$ and (because at the time of the pre-condition T is integer) $Q{=}q\ \underline{\textbf{for}}(\lceil T\rceil - T)\ \ \underline{\textbf{until}}\ \ Q{\neq}q$. So the law applies and

$$Wait \sqsubseteq \texttt{wait on Q}$$

This gives the oscillator the final shape

```
process [Q : out]
begin
  Q ⇐  transport ¬Q after 1;                (s3)
  wait on Q;
end
```

and this code is correct by construction. It satsifies the initial specification (s_0),

2.4. Miscellania

One extra rule of refinement has been implicit in the above reasoning. A specification may be satisfied by a stronger specification:

> **Law 5** *(Weakening) A pre-condition can be strengthened or a during- or post-condition can be weakened or a side-condition can be strengthened without affecting the validity of a refinement step. I.e., if:*
>
> $$(PRE \rightarrow pre) \wedge (dur \rightarrow DUR) \wedge (post \rightarrow POST) \wedge (ENV \rightarrow env)$$
>
> *then*
>
> $$\frac{[{}^{\backprime}pre \mid dur \mid post'\ \texttt{where}\ env] \sqsubseteq a}{[{}^{\backprime}PRE \mid DUR \mid POST'\ \texttt{where}\ ENV] \sqsubseteq a}$$

3. Further example

In this section a more substantial piece of hardware design is considered: a single-element *RS flip-flop*. It is not straightforward to write a general asynchronous specification for this component so we will assume a unit-time clock and write

$$\begin{array}{rcl} \neg\ Q & = & \circledbullet\,(\neg\ S\ \underline{\textbf{since}}\ R) \\ \neg\ \overline{Q} & = & \circledbullet\,(\neg\ R\ \underline{\textbf{since}}\ S) \end{array}$$

for the specification. I.e., the signal Q will be low so long as there had been, up until one unit of time ago, no set signal S received since the reset signal R was last high. And vice versa for $\overline{Q}$. The symbol " $\circledbullet$ " means "one unit of time ago". The "<u>**since**</u>" combinator has its natural meaning. The subtleties of what happens if the pior condition is never obtained are not of interest, because we will straight away pass (via *Weakening*) to a refinement that we know gives the "right" solution in unit-time (under certain assumptions on the input signals), viz:

FlipFlop

${}^{\backprime}T = 0$	$\neg\ Q = \circledbullet\,(R \vee \overline{Q})$ $\neg\ \overline{Q} = \circledbullet\,(S \vee Q)$	*false*
${}^{\backprime}T \leq T \leq T'$		

The least solution to this mutual recursion is:

$$\begin{array}{l} \neg\ Q = \circledbullet\,R \vee \circledbullet^{2}\neg\ S \wedge \circledbullet^{3}R \vee \ldots \\ \neg\ \overline{Q} = \circledbullet\,S \vee \circledbullet^{2}\neg\ R \wedge \circledbullet^{3}S \vee \ldots \end{array}$$

which satisfies the original specification provided that *each set or reset command lasts at least through two consecutive clock points.* These are external conditions on the environment, and they are acceptable. The *FlipFlop* specification can then be split into two concurrent parts (here the ${}^{\backprime}T \leq t \leq T'$ is suppressed for legibility):

Flip

$$`T = 0 \mid \neg\, Q = \;\odot\, (R \vee \overline{Q}) \mid \mathit{false}$$

$$\underline{\mathbf{always}}(\neg\, \overline{Q} = \;\odot\, (S \vee Q))\, \underline{\mathbf{since}}\; T = 0$$

Flop

$$`T = 0 \mid \neg\, \overline{Q} = \;\odot\, (S \vee Q) \mid \mathit{false}$$

$$\underline{\mathbf{always}} \neg\, Q = \;\odot\, (R \vee \overline{Q})\, \underline{\mathbf{since}}\; T = 0$$

and in each of these, the others output signal appears as an input and the other specification appears as an environmental condition. The appropriate formal law appears below:

Law 6 *(Parallel)*

$$\frac{[`x \mid y_1 \mid z' \;\texttt{where}\; y_2\, \underline{\mathbf{since}}\, x\, \underline{\mathbf{until}}\, z] \sqsubseteq a \qquad [`x \mid y_2 \mid z' \;\texttt{where}\; y_1\, \underline{\mathbf{since}}\, x\, \underline{\mathbf{until}}\, z] \sqsubseteq b}{[`x \mid y_1 \wedge y_2 \mid z'] \sqsubseteq a \,||\, b}$$

We strengthen the specifications again by allowing each to hold in quite generic environments, not merely the environment provided by the other (*Weakening*):

FLIP

$$`T = 0 \mid \neg\, Q = \;\odot\, (R \vee \overline{Q}) \mid \mathit{false}$$

$$`T \leq T \leq T'$$

FLOP

$$`T = 0 \mid \neg\, \overline{Q} = \;\odot\, (S \vee Q) \mid \mathit{false}$$

$$`T \leq T \leq T'$$

with

$$\mathit{Flip} \sqsubseteq \mathit{FLIP} \qquad \mathit{Flop} \sqsubseteq \mathit{FLOP}$$

and

$$\mathit{FlipFlop} \sqsubseteq \mathit{Flip} \,||\, \mathit{Flop} \sqsubseteq \mathit{FLIP} \,||\, \mathit{FLOP}$$

From here the specifications can be refined using the laws set out in the previous section. We obtain (abbreviating the VHDL expression syntax):

FLIP $\sqsubseteq$

```
process [Q:out,R:in]
begin
   Q ⇐ transport ¬R∧¬Q̄ after 1;
   wait on (R, Q̄)
end
```

FLOP $\sqsubseteq$

```
process [Q̄:out,S:in]
begin
   Q̄ ⇐ transport ¬S∧¬Q after 1;
   wait on (S, Q)
end
```

which pair of codes, by construction, jointly satisfies the *FlipFlop* specification at the head of this section.

4. Conclusion

A formal specification and refinement calculus for unit-delay behavioural VHDL has been set out here. A specification defines up an activation pre-condition, an invariant and a post-condition. These hold true respectively on entry to the specification, while it is running, and on exit from the process. Non-termination can be expressed through a *false* post-condition. A set of six formal rules governs refinement of these specifications to VHDL code through a series of mixed code-and-specification intermediates. The resulting VHDL code is "correct by construction".

In future it is hoped that graphical notations such as timing diagrams can be appropriated for use as the specification format.

References

[1] J.P. Bowen, He Jifeng and I. Page. Hardware Compilation. In J.P. Bowen (ed.), *Towards Verified Systems*, Elsevier Science, Real-Time Safety Critical Systems series, volume 2, Chapter 10, pp 193–207, 1994.

[2] P.T. Breuer, L. Sánchez and C. Delgado Kloos. A Simple Denotational Semantics, Proof Theory and a Validation Condition Generator for VHDL. *Journal of Formal Methods for System Design*, 7(1&2):27–51, 1995.

[3] F. Christian. Correct and robust programs. *IEEE Trans. on Software Engineering*, 10:163–174, March 1994.

[4] E.M. Clarke, E.A. Emerson and A.P. Sistla. Automatic Verification of Finite-state Concurrent Systems using Temporal Logic Specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.

[5] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman and M. Trachtenbrot. Statemate: a working Environment for the development of complex reactive systems. *IEEE Transactions of Software Engineering*, 16:403–414, 1990.

[6] He Jifeng and J.P. Bowen. Time Interval Semantics and Implementation of a Real-Time Programming Language. *Proc. Fourth Euromicro Workshop on Real-Time Systems*, IEEE Computer Society Press, pp 110–115, 1992.

[7] S. King and C. Morgan. Exits in the refinement calculus. *Formal Aspects of Computing*, 7:54–76, 1995.

[8] Carroll Morgan. *Programming from Specifications.* Prentice Hall International Series in Computer Science, 1994.

[9] R. Schlör and W. Damm. Specification and verification of system-level hardware designs using timing diagrams, *The European Conference on Design Automation with the European Event in ASIC Design*, pp 518–524, 1993.

[10] J.M. Spivey. Specifying a Real-Time Kernel, *IEEE Software*, 7(5):21–28, September 1990.

[11] J.M. Spivey. *The Z Notation: A reference manual*, Prentice-Hall International Series in Computer Science, 2nd edition, 1992.