A Refinement Calculus for the Synthesis of Verified Hardware Descriptions in VHDL

PETER T. BREUER, CARLOS DELGADO KLOOS, ANDRÉS MARÍN LÓPEZ, and NATIVIDAD MARTÍNEZ MADRID
Universidad Carlos III de Madrid
and
LUIS SÁNCHEZ FERNÁNDEZ
Universidad Politécnica de Madrid

A formal refinement calculus targeted at system-level descriptions in the IEEE standard hardware description language VHDL is described here. Refinement can be used to develop hardware description code that is ‘‘correct by construction.” The calculus is closely related to a Hoare-style programming logic for VHDL and real-time systems in general. That logic and a semantics for a core subset of VHDL are described. The programming logic and the associated refinement calculus are shown to be *complete*. This means that if there is a code that can be shown to implement a given specification, then it will be derivable from the specification via the calculus.

Categories and Subject Descriptors: B.1.2 [**Hardware**]: Control Structures and Microprogramming—*formal models*; B.1.4 [**Hardware**]: Control Structures and Microprogramming—*languages and compilers*; D.3.3 [**Programming Languages**]: Language Semantics

General Terms: Design, Languages, Theory, Verification
Additional Key Words and Phrases: Denotational semantics, digital circuits, formal verification, program logic, refinement, timed logic, VHDL

1. INTRODUCTION

This article sets out theory to support the engineering of verified descriptions of hardware systems written in the IEEE standard hardware description language VHDL [IEEE 1988]. The *refinement engineering process* starts with a set of loose but formally expressed constraints on the desired system behavior. At every step the

The work reported in this article has been partially supported by the EuroForm network (2/ERB4050PL921826) sponsored by the Human Capital and Mobility programme of the CEC and the NADA project (ESPRIT Working Group 8533).

Authors' addresses: P.T. Breuer, C. Delgado Kloos, A. Marín López, and N. Martínez Madrid, Area de Ingeniería Telemática, Departmento Ingeniería, Universidad Carlos III de Madrid, Butarque 15, E–28911 Leganés, Madrid, Spain; email: {ptb; cdk; amarin; nmadrid}@it.uc3m.es; L. Sánchez Fernandez, Departmento Ingeniería de Sistemas Telemáticos, ETSI Telecomunicación, Universidad Politécnica de Madrid, Ciudad Universitaria, E–28040 Madrid, Spain; email: lsanchez@dit.upm.es.

designer chooses a point within the specification and a refinement transformation to apply to it, and a new, more code-like, specification is calculated. The final result is a VHDL description code that satisfies the initial specification. It is ‘‘correct by construction” because each step is sound in a formal sense. The set of refinement transformations comprises a *refinement calculus* [Morgan 1994]. The calculus is complete, which means that if a satisfactory VHDL code exists, then it can be obtained from the initial specification by an appropriate sequence of refinements.

The idea of a refinement calculus is not new in the context of formal methods for software system development (for example, see Morris [1987], Back and Sere [1991], and Morgan [1994]), but this technique is new in the context of the hardware description language VHDL. Its development has had to wait upon formal semantics and logics for VHDL (e.g., Salem and Borrione [1992], Déharbe and Borrione [1995], and Breuer et al. [1995] and other papers in Borrione [1995]), and it is in the past year or so that the necessary basis has been set up by those working in the field [Gordon 1995]. It is the formal semantics of VHDL collectively given in Delgado Kloos and Breuer [1995] that is elaborated here.

The following are the elements presented in this article:

(1) a refinement calculus for VHDL;

(2) a specification formalism for real-time systems;

(3) a scheduling semantics for temporal logic;

(4) a descriptive style for real-time programs that uses three logical assertions instead of two, as explained below.

The extra assertion states what is true *during* program execution, in addition to the usual program *precondition* and *postcondition.* The style derives from attempts to phrase the semantics of software programs with interrupts [Christian 1994] and a recent description of a refinement calculus for programs with multiple exits in the control flow graph [King and Morgan 1995]. There is an associated specification calculus [Breuer et al. 1996a]. The extra *during*-condition is related to the guarantee part of *rely/guarantee* approaches to concurrent processes [Jones 1983].

Another approach to the design of hardware via refinement is to be found in the LAMBDA theorem-proving environment [Mayger et al. 1991], and there is related work of Mahony and Hayes [1991] giving a theory of refinement aimed at dataflow-style languages. Both approaches use real functions of time with no future aspect as the model and are sited mainly on the specification side of the implementation/specification divide. The target formalism is declarative.

A similar approach to LAMBDA is used in many other systems based on higher-order logic, such as HOL [Gordon and Melham 1993]. The general technique is to express the desired circuit succinctly using higher-order circuit combinators and then prove that the functionality is as required (e.g., see Kropf et al. [1994]).

We only know of one other temporal logic-based approach (an interval logic) to VHDL. This is the work of Wilsey [1992]. But at the present time that research is directed toward static analysis of the time properties of VHDL codes rather than toward a full dynamic semantics or a design calculus. The analysis of more classical high-level languages with respect to time is treated, for example, in Shaw [1989] and Kopetz et al. [1991].

Temporal logics have typically been used in model checking approaches to the verification of medium-sized hardware circuits, but their application to VHDL has had to wait upon complete and tractable formal semantics for the language. And although finite-state machine approaches to the nondelay parts of VHDL codes have been pursued for some while (e.g., Borrione et al. [1992]), the first effort at a more complete operational semantics [van Tassel 1990; 1993] is still recent.

There is a semiclassical approach to the refinement of general software systems in which real time, as well as functional, behavior is specified. It consists of adding a single reserved variable T to an existing method for untimed systems. An excellent example is the work of Fidge [1993], based on refinement of specifications in Z [Wordsworth 1992] (but not targeted at any coding language). The technique occurs very often in the literature.

Refinement techniques such as the above should be contrasted with techniques of *program transformation*, particularly for hardware (e.g., Bose and Johnson [1993]). In these styles a correct functionality is first written down in a functional or logic programming language, and then an efficient coding is derived via a suite of semantics-preserving transformations. Alternatively, a proof of equivalence is derived "between the design and the circuit implementation" [Johnson et al. 1994]. Refinement, on the other hand, changes functionality. It proceeds from a specification that can be satisfied by many different functionalities toward a code with one single final functionality.

The ease or difficulty of working with the formalisms set out here is not the subject of this article. It is certainly the case that particular programs may prove amenable to particular verification methods such as symbolic evaluation or model checking, but it is our practical experience that verifying the correctness of any existing VHDL description code is always difficult. The code usually contains implicit assumptions about the operating environment or mode of use that have not been registered in the deliverable, but which would be very helpful to a formal verification. Without knowledge of those constraints, the code may appear to be incorrect. Moreover, the way the author thought about constructing the code has usually not been recorded. Without that information, the thought process has to be reconstructed in order to partition the verification problem appropriately. Further, it is hardly ever the case that a high-level formal specification for a hardware design is available. The specification may be recorded as "a D flip-flop" or "an arithmetic logic unit" along with tables of input and output values or descriptions of point-to-point functionality plus minimum and maximum delays. So a specification in an appropriate format has to be constructed before formal verification of the design can begin.

Given that, the idea of producing a design from a specification via a series of formally verified steps overcomes some obstacles. It introduces in their place the problem of constructing the formal specification and then reasoning from it within the refinement calculus and its associated logic. Significantly, it returns to the user a documented series of design steps leading to the final hardware design. Each step consists of a specification and its partial refinement.

The layout of this article is as follows. We briefly describe the essential features of VHDL in Section 2 and give a denotational semantics for the language in Section 3. We recapitulate a formal semantics of VHDL first set out in Breuer et al. [1995]

and which is here rephrased. We introduce our specification format and semantics in Section 4, and formal refinement is introduced in Section 5. A worked example illustrates the refinement rules with a statement-by-statement derivation of a simple piece of hardware in Section 6. Section 7 deals with parallelism in particular through another worked example. Finally, Section 8 deals with theory, introducing an axiomatic logic with which completeness of the refinement calculus is proved.

2. BRIEF DESCRIPTION OF VHDL

The IEEE standard hardware description language VHDL has traditionally been seen as rather a baroque language, but leaving aside the impressive array of scoping, packaging, and typing constructs, the sharp end consists of a procedural sublanguage with the imperative flavor of Pascal or C, plus a parallelism component.

A VHDL "model" of a hardware system consists of a set of *process* definitions, each containing purely procedural code.

The concept is that hardware components are represented by processes running in parallel and communicating via *signals*. Layered structure in a model is represented through the use of code-generating macros. When these are unfolded they result in an *elaborated model*. This article deals with the semantics of elaborated VHDL. Note that parallelism here can be only one layer deep, because processes do not contain other processes.

The procedural code in process bodies can contain `while` loops, `if` branches, and so on, as well as assignments to local variables. The code inside each process body loops continuously. In summary, the procedural code looks entirely standard to the eye, apart from two special kinds of statement, signal assignment and wait statements, that are described immediately below:

(1) Signal assignment:

```
C <= transport (not C) after dly
```

The statement above schedules an assignment in `dly` time units time of the logical negation of the current value of signal C to signal C again. If the delay were zero, the assignment would be called *delta-delayed*.[1]

There is a second form of signal assignment in VHDL. The *inertial* assignment syntax lacks the `transport` modifier. Its semantics is similar, and it will not be dealt with here. The treatment is simply analogous to that for transport assignment.

(2) The explicit `wait` statement:

```
wait on C
```

This statement blocks until such time as a *level transition* occurs on signal `C`. The following is the general form:

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

Any two of the three parts are optional.

[1] Delta-delayed assignments and "zero-delay" waits are treated as being infinitesimally delayed in Breuer et al. [1996b], i.e., delayed by some nonzero but negligible positive amount.

The VHDL standard's definition of `wait` statement semantics is in terms of process scheduling in the completed VHDL model. The following is a précis. The *signals* are the list of signals to which the `wait` statement is sensitive. If the `until` part is included, then each time a level transition occurs in a signal included in the list the *test* is evaluated. If the result is `true`, the process will resume; else it will remain suspended. Entry into a `wait for` statement causes the *time* to be evaluated to determine the default timeout interval. The process will resume immediately after the timeout interval expires if no transitions occur on the listed signals.

While it makes sense, the definition is closely bound to an operational view. Just what does a `wait` statement do? The answer is not wholly contained in the paragraph above. It is necessary to understand the definition in the broader context of the way VHDL processes are scheduled, according to the standard.

A standard VHDL simulation "runs" each process in a VHDL description until it becomes blocked in a `wait` statement. From the point of view of the modeled hardware, zero time elapses to this point. The simulation time is then advanced to the next pending assignment time, and the assignment is executed. The conditions of each `wait` statement are then checked, and if any are satisfied the appropriate process bodies are executed (in zero simulation time) until they all become blocked again. Then simulation time is advanced again, and so on.

The operational and holistic nature of the above descriptions is evident. Beginning students can take a very long time to gain facility with the language.

We concentrate on a proper subset of VHDL here. The following combinators are used, abbreviating the syntax of standard VHDL: `;` (sequence), $||$ (parallel composition), $\Leftarrow$ (transport signal assignment) and `wait`, `if` (conditional branch), `while` (loop), and `process` (encapsulation). The following restrictions apply to the subset:

(1) There are no local variables. Signals and expressions are used in their place. This is emphatically *not* a restriction of the technique, but a simplification for the presentation here.[2] See, for example, Morgan [1994] for a standard treatment of local variables in refinement calculi.

(2) The combination "`<= transport`" will be abbreviated by "$\Leftarrow$." There is no confusion with inertial assignment because only the transport form of assignment is treated here.

(3) The delay in signal assignments (and in `wait for` statements) may not be smaller than some specific nonzero lower bound. This is to prevent signal assignments and `wait` statements together being used to implement essentially recursive calculations in negligible ("delta") time. Recursive functions can be obtained by using calls to external library functions instead.

(4) The `process` combinator forms a process from a (possibly compound) statement. Notionally, processes are continuously looping statements. In distinction to VHDL proper, the *output* signals of a process are listed in the header:

[2] Local variables can always be eliminated from full VHDL: references to a variable which does not persist across time can be substituted by appropriate symbolic combinations of references to the input signals. Variables which persist across time can be implemented via signals; the process can signal to itself (in the future) the nominal contents of its state (in the present).

```
Statement ::= Id ⇐ Expression after Delay
            | wait on Ids
            | null
            | Statement ; Statement
            | if Expr then Statement else Statement
            | while Expr do Statement

Process   ::= process [ Ids ] begin Statement end
            | Process || Process
```

Fig. 1. The concrete syntax of the kernel subset of VHDL hardware descriptions treated here. See text for additional constraints.

```
process [sig_1, ..., sig_K]
begin
  statement_1^- ;
  ...
  statement_N ;
end
```

Each sig_i declares a signal wire over which the process has (sole) output rights. Input signals are not listed. VHDL proper uses round brackets here to denote a *sensitivity list*, and we use square brackets to denote a *output signal list.*

(5) *Resolution functions*, which in full VHDL are used to combine the outputs of several processes writing to the same wire, are not treated.[3]

(6) The *concurrent statements* of full VHDL are not treated here. They have the semantics of a process with only the statement in its body.

(7) Only the `wait on` *signals* form of `wait` is considered here. Any one of the simple forms of `wait` can generate the full range of wait semantics, and this decision is not significant.[4]

(8) All `while` loops must contain a `wait` statement in every branch through the loop. This is to prevent computations of unbounded complexity occurring in bounded time during the simulation.

The Backus-Naur Form (BNF) for the concrete syntax of the kernel of VHDL treated here is given in Figure 1. The set of allowed codes has full functional coverage, and so any functionality allowed by a specification may be obtained using one of these codes. That one code may be refined to via the refinement calculus.

The next section lays groundwork for the rest of the article by describing the denotational semantics of the subset of VHDL above. The denotation follows the

[3] Resolution functions can be eliminated from any VHDL code by everywhere replacing references to the resolved signal by the appropriate combinatorial expression (of the contributing signals).

[4] For example, a `wait for` *time* can be emulated by assigning an event on a dedicated timing signal to mature in *time* units. Then a `wait on` the signal will resume after *time.* A `wait until` *cond* can be emulated with a `while` *cond* loop enclosing a `wait for` or `wait on`, and so on.

pattern laid down in Breuer et al. [1995] except in that it does not assume a discrete time domain.

3. DENOTATIONAL SEMANTICS OF VHDL

Hardware descriptions in VHDL and hardware specifications have denotations in the same domain. Each denotation can be thought of as a relation. A "state of the system" before a process runs is related to a state of the system after or during the run. A process will denote a many-one relation on the signals that it controls, but will denote a many-many relation over at least its external signals.

The two "states of the system" that a process relates each have an extension forward and backward in time. Forward references are to a projected schedule rather than an inevitable fact. Every process starts off with a view of the system that includes the history it has seen so far, the situation at the moment, and a tentative schedule for what is to happen in the future. The whole sequence comprises a *world line*, borrowing a term from general relativity. A world line contains the information that in the VHDL community is known as a *driver set* (a schedule of pending changes). The world line contains in its future part exactly the sequence of projected states that will result from the application of the pending changes queued in the corresponding VHDL driver set.

During or after a process run, the world line that a process sees may differ in two ways from the initial world line that it saw:

(1) the historical part of the world line will be conserved and extended;
(2) the "future" (i.e., scheduled) part of the world line may have changed, and it will be shorter, in the sense of now extending over a smaller time range.

The picture here is that a process connects two different world lines. It can be imagined as an infinitely long two-dimensional strip surface bounded by the two world lines. The two world lines are equal on the tail of the strip, their common history.

To that picture one element has to be added. On the surface of the strip lies the trajectory traveled by a current time pointer as it changes between its bindings in the initial and final world lines. Figure 2(a) shows the picture that is appropriate to the `wait on C` statement, and Figure 2(b) shows the picture for a signal assignment. Figure 3 shows how the pictures for sequential statements are combined. The circles in the pictures represent successive states along the world lines related by the process. Although drawn as discrete here, they may form a continuous sequence, depending on the underlying model of time.

Formally, a world line is a time-indexed sequence of states (*worlds*). A state is a partial function from signal identifiers to values:

$$\begin{aligned} WorldLine &= Time \rightarrow State \\ State &= Id \nrightarrow Value \end{aligned}$$

$Value$ is any fixed but sufficiently rich domain. It includes at least the integers.

A process denotes a relation between an "initial" world line and a current time and a "later" world line and current time. Here $A \leftrightarrow B$ is the type of relations with domain A and codomain B:

$$Semantics = (WorldLine, Time) \leftrightarrow (WorldLine, Time)$$

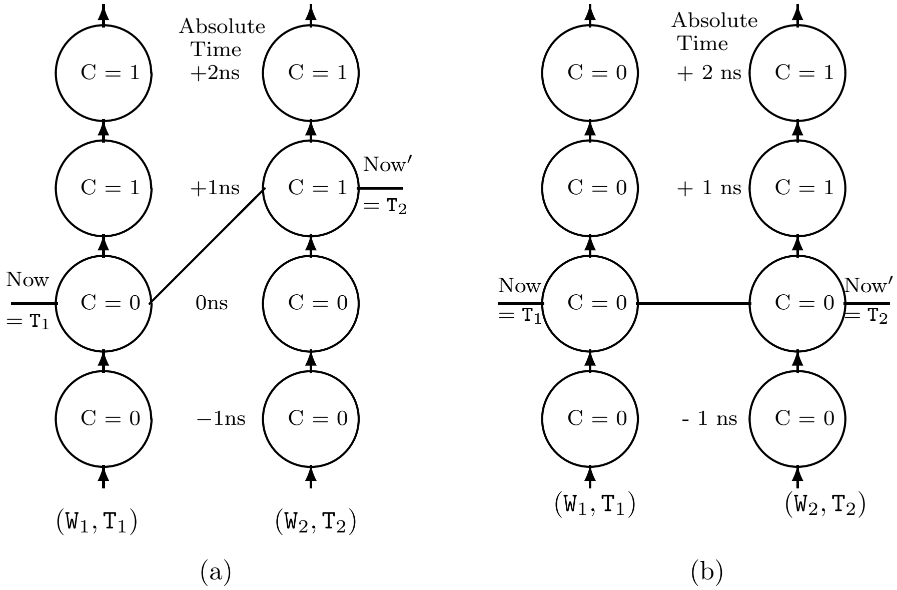


Fig. 2. (a) illustrating the semantics of the `wait on C` statement. No change in historic or planned states, but a forward shift in the current time pointer to the first state in which `C` has changed. (b) the semantics of the signal assignment `C`$\Leftarrow$`not C after 1ns`. The assignment writes into every future state and the current time pointer is unchanged.

There are *two* variant semantics to be considered:

(1) the semantics of running a *code* to *termination* in a given environment ρ, a list of signals that may be affected by the code:

$$\mathcal{S}\rho[\![code]\!] \ : \ Semantics$$

(2) for a given environment ρ, the semantics of running a *code* on to some point at which it is still running:

$$\mathcal{P}\rho[\![code]\!] \ : \ Semantics$$

The latter is called the *suspension* semantics. The name derives from the idea that one may have pressed "control-Z" (a Unix shell command line keypress that generates a stop signal to a running process) in order to examine the world line and current time pointer pair.

The environment ρ is actually used to determine exactly which signals must remain constant during a wait statement execution. Suspension semantics is necessary in order to reason about running processes in VHDL. These never terminate, so there is no sense in considering their termination semantics.

Table I contains a detailed formal termination and suspension denotation for each VHDL code construct. The termination semantics is as illustrated in the figures

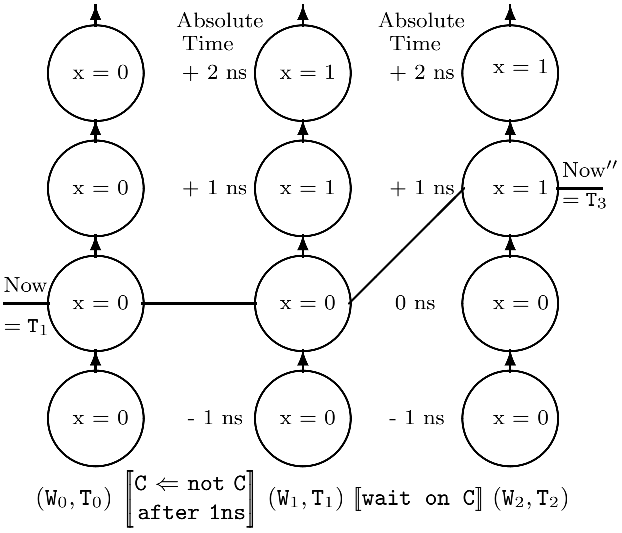


Fig. 3. Composing two statements in sequence. They compose as relations.

here. Transport assignments change the future schedule but not the current time pointer. `Wait` statements change the time pointer but not the current schedule. The suspension semantics differs in the following ways:

(1) a transport assignment statement has the empty relation as its suspension semantics. It takes no time to execute and cannot be suspended at all;

(2) a `wait` statement may be suspended, but the time point at suspension must be strictly before the time point at which the statement is due to terminate;

(3) a `while` loop may suspend only in the loop body; a conditional may suspend only in one or another branch; a `null` statement may not suspend;

(4) a sequence of two statements may suspend *either* in the first *or* in the second of the two as described below.

The most revealing difference lies in the semantics of sequential statement composition. The termination semantics for a sequence $code_1; code_2$ is just relational composition:

$$\mathcal{S}\rho[\![code_1; code_2]\!] = \mathcal{S}\rho[\![code_1]\!] \fatsemi \mathcal{S}\rho[\![code_2]\!]$$

But the suspension semantics represents the idea that (1) the sequence may be suspended either in the first statement or (2) the first statement may run to completion, and then the suspension may occur in the second:

$$\mathcal{P}\rho[\![code_1; code_2]\!] = \mathcal{P}\rho[\![code_1]\!] \cup \mathcal{S}\rho[\![code_1]\!] \fatsemi \mathcal{P}\rho[\![code_2]\!]$$

For both semantics, parallelism is represented by the intersection of relations.

In the next section, the termination and suspension semantics of specifications will be described.

4. SPECIFICATION

Consider the design of an oscillator `a` with a half-period equal to the basic system clock cycle time. During even time intervals, when $T \in [2n, 2n+1)$, it should emit

Table I. Formal Definitions for the Termination and Suspension Semantics of VHDL

Syntax R	Termination Semantics ${}^{\backprime}W\,{}^{\backprime}T\ \mathcal{S}\rho\llbracket R\rrbracket\ W'T'$
$x{\Leftarrow}y\ \texttt{after}\ \tau$	${}^{\backprime}T{=}T' \wedge \forall t{\geq}{}^{\backprime}T{+}\tau \bullet$ $W't\vert_\rho{=}{}^{\backprime}Wt\vert_\rho{\oplus}\{(x,\ {}^{\backprime}W\,{}^{\backprime}Ty)\} \wedge \forall t{<}{}^{\backprime}T{+}\tau \bullet W't\vert_\rho{=}{}^{\backprime}Wt\vert_\rho$
$\texttt{wait on}\ x$	${}^{\backprime}WT'x \neq {}^{\backprime}W\,{}^{\backprime}Tx \wedge \forall t\vert\,{}^{\backprime}T{\leq}t{<}T' \bullet {}^{\backprime}Wtx = {}^{\backprime}W\,{}^{\backprime}Tx \wedge {}^{\backprime}W\vert_\rho{=}W'\vert_\rho$
$\texttt{if}\ x\ \texttt{then}\ R_1\ \texttt{else}\ R_2$	$\mathit{if}\ {}^{\backprime}W\,{}^{\backprime}Tx\ \mathit{then}\ {}^{\backprime}W\,{}^{\backprime}T\mathcal{S}\rho\llbracket R_1\rrbracket W'T'\ \mathit{else}\ {}^{\backprime}W\,{}^{\backprime}T\mathcal{S}\rho\llbracket R_2\rrbracket W'T'$
$R_1; R_2$	$\exists W_mT_m.\,{}^{\backprime}W\,{}^{\backprime}T\mathcal{S}\rho\llbracket R_1\rrbracket W_mT_m\mathcal{S}\rho\llbracket R_2\rrbracket W'T'$
$\texttt{while}\ x\ \texttt{do}\ R$	$\mathit{if}\ \neg{}^{\backprime}W\,{}^{\backprime}Tx\ \mathit{then}\ {}^{\backprime}W\,{}^{\backprime}T\mathcal{S}\rho\llbracket\texttt{null}\rrbracket W'T'$ $\mathit{else}\ {}^{\backprime}W\,{}^{\backprime}T\mathcal{S}\rho\llbracket R;\texttt{while}\ x\ \texttt{do}\ R\rrbracket W'T'$
$\texttt{null}$	${}^{\backprime}T = T' \wedge {}^{\backprime}W\vert_\rho = W'\vert_\rho$
$\texttt{process [}\sigma\texttt{:out]}$ $\texttt{begin}\ R\ \texttt{end}$	false
$R_1 \Vert R_2$	${}^{\backprime}W\,{}^{\backprime}T\mathcal{S}\rho\llbracket R_1\rrbracket W'T' \wedge {}^{\backprime}W\,{}^{\backprime}T\mathcal{S}\rho\llbracket R_2\rrbracket W'T'$
Syntax R	**Suspension Semantics ${}^{\backprime}W\,{}^{\backprime}T\ \mathcal{P}\rho\llbracket R\rrbracket\ WT$**
$x{\Leftarrow}y\ \texttt{after}\ \tau$	false
$\texttt{wait on}\ x$	$\forall t\vert\,{}^{\backprime}T \leq t \leq T \bullet {}^{\backprime}Wtx{=}{}^{\backprime}W\,{}^{\backprime}Tx \wedge {}^{\backprime}W\vert_\rho{=}W\vert_\rho$
$\texttt{if}\ x\ \texttt{then}\ R_1\ \texttt{else}\ R_2$	$\mathit{if}\ {}^{\backprime}W\,{}^{\backprime}Tx\ \mathit{then}\ {}^{\backprime}W\,{}^{\backprime}T\mathcal{P}\rho\llbracket R_1\rrbracket WT\ \mathit{else}\ {}^{\backprime}W\,{}^{\backprime}T\mathcal{P}\rho\llbracket R_2\rrbracket WT$
$R_1; R_2$	$\exists W_mT_m.\,{}^{\backprime}W\,{}^{\backprime}T\mathcal{S}\rho\llbracket R_1\rrbracket W_mT_m\mathcal{P}\rho\llbracket R_2\rrbracket WT \vee {}^{\backprime}W\,{}^{\backprime}T\mathcal{P}\rho\llbracket R_1\rrbracket WT$
$\texttt{while}\ x\ \texttt{do}\ R$	$\mathit{if}\ \neg{}^{\backprime}W\,{}^{\backprime}Tx\ \mathit{then\ false\ else}\ {}^{\backprime}W\,{}^{\backprime}T\mathcal{P}\rho\llbracket R;\texttt{while}\ x\ \texttt{do}\ R\rrbracket WT$
$\texttt{null}$	false
$\texttt{process [}\sigma\texttt{:out]}$ $\texttt{begin}\ R\ \texttt{end}$	${}^{\backprime}W\,{}^{\backprime}T\mathcal{P}\rho\sigma\llbracket \rho{:=}0;\ \rho{\Leftarrow}0\ \texttt{after}\ 0;\texttt{while}\ \mathit{true}\ \texttt{do}\ R\rrbracket WT$
$R_1 \Vert R_2$	${}^{\backprime}W\,{}^{\backprime}T\mathcal{S}\rho\llbracket R_1\rrbracket WT \wedge {}^{\backprime}W\,{}^{\backprime}T\mathcal{S}\rho\llbracket R_2\rrbracket WT$

The set ρ names the output signals of the controlling process, and, for each code R, $\mathcal{S}\rho\llbracket R\rrbracket$ and $\mathcal{P}\rho\llbracket R\rrbracket$ describe a relation between (*WorldLine*, *Time*) pairs.

a zero on signal `Q`, and during odd time intervals $T \in [2n+1, 2n+2)$ it should emit a one. That is, the value Q on `Q` is

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

The startup time will be denoted ${}^{\backprime}T$, with a preceding "antiprime." We want ${}^{\backprime}T = 0$, and at this time no conditions on the state will be imposed. Contrariwise, no constraints will be imposed on the times T at which the process output `Q` may be inspected, but the value on `Q` will be constrained to take the value $Q = \lfloor T \,\mathbf{mod}\, 2\rfloor$ then. The time at which the oscillator terminates will be denoted T', and we want $T' = \infty$.

The startup time requirement ${}^{\backprime}T = 0$ and the requirement $Q = \lfloor T \,\mathbf{mod}\, 2\rfloor$ at the suspension times are the *pre-* and *during-conditions* respectively. A suitable *postcondition* for the oscillator is *false*, because we do not want the oscillator to terminate. The three conditions are written as a *specification triple*:

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

The leading Q names the unique signal controlled by this process. An antiprime distinguished the variables in the precondition; a prime distinguishes the variables in the postcondition. By convention, a prime (antiprime) on a formula means that

Table II. The Interpretation of Logic within a World Line W at Time t

Predicate Syntax	Predicate Semantics
Predicate	
$::= \mathit{Predicate}\ \{\wedge \mid \vee \mid \rightarrow\}\ \mathit{Predicate}$	$Wt \models p \wedge q \quad \text{iff}\ Wt \models p \wedge Wt \models q$
$\mid\ \neg\ \mathit{Predicate}$	$Wt \models \neg p \quad \text{iff}\ \neg(Wt \models p)$
$\mid\ \forall \mathit{Id} \bullet \mathit{Predicate} \mid \exists \mathit{Id} \bullet \mathit{Predicate}$	$Wt \models \exists X \bullet p\ \text{iff}\ \exists x \bullet Wt \models p[x/X]$
	$Wt \models \forall X \bullet p\ \text{iff}\ \forall x \bullet Wt \models p[x/X]$
$\mid\ \{\odot \mid \underline{\Box} \mid \underline{\Diamond}\}\ \mathit{Predicate}$	$Wt \models \underline{\Box}\, p \quad \text{iff}\ \forall t' \geq t \bullet Wt' \models p$
	$Wt \models \underline{\Diamond} p \quad \text{iff}\ \exists t' \geq t \bullet Wt' \models p$
	$Wt \models \odot^{\tau} p \quad \text{iff}\ W(t+\tau) \models p$
$\mid\ \mathit{Predicate}\ \underline{\texttt{for}}\ \mathit{Time}$	$Wt \models p\, \underline{\texttt{for}}\, \tau\ \text{iff}\ \forall t' \mid t \leq t' < t+\tau \bullet Wt' \models p$
$\mid\ \texttt{true} \mid \texttt{false}$	$Wt \models \texttt{true} \quad \neg(Wt \models \texttt{false})$
$\mid\ \mathit{Expression}\ \{= \mid \neq \mid < \mid >\}\ \mathit{Expression}$	$Wt \models \texttt{X} = \texttt{Y} \quad \text{iff}\ Wt\texttt{X} = Wt\texttt{Y}$
	$Wt \models \texttt{T} = t' \quad \text{iff}\ t = t'$
$\mid\ \mathit{Predicate}\ \{\underline{\texttt{until}} \mid \underline{\texttt{then}}\}\ \mathit{Predicate}$	$p\, \underline{\textbf{until}}\, q \iff \underline{\Box}\, p \vee \exists \tau \geq 0 \mid \odot^{\tau} q \bullet p\, \underline{\textbf{for}}\, \tau$
	$p\, \underline{\textbf{then}}\, q \iff p\, \underline{\textbf{until}}(q \wedge \neg p)$

Expression Syntax	Expression Semantics
Expression	
$::= \odot^{\mathit{Time}} \mathit{Expression}$	$Wt \models \odot^{\tau} e \quad = W(t+\tau) \models e$
$\mid\ \mathit{Val} \mid \mathit{Id}$	$Wt \models k = k \quad Wt \models \texttt{X} = Wt\texttt{X} \quad Wt \models \texttt{T} = t$
$\mid\ -\ \mathit{Expression}$	$Wt \models -e \quad = -(Wt \models e)$
$\mid\ \mathit{Expression}\ \{\texttt{+} \mid \texttt{-} \mid \texttt{/} \mid \texttt{*}\}\ \mathit{Expression}$	$Wt \models e_1 + e_2 = Wt \models e_1 + Wt \models e_2$
$\mid\ \mathit{Predicate}\texttt{?}\mathit{Expression}\texttt{:}\mathit{Expression}$	$Wt \models p?e_1{:}e_2 = Wt \models p?Wt \models e_1{:}Wt \models e_2$

a prime (antiprime) is added to all the free variables in the formula.

A specification triple $\rho[{}^{\backprime}\mathit{pre} \mid \mathit{dur} \mid \mathit{post}']$ gives rise to termination and suspension semantics as follows. The two pairs $({}^{\backprime}\mathit{pre}, \mathit{post}')$ and $({}^{\backprime}\mathit{pre}, \mathit{dur})$ respectively generate the two semantics. Let *modeling* of a condition p by a pair consisting of a world line and current time pointer Wt (written $Wt \models p$) be as defined in Table II. The termination and suspension semantics of a specification triple are the sets of pairs:

$$\mathcal{S}\rho[\![{}^{\backprime}\mathit{pre} \mid \mathit{dur} \mid \mathit{post}']\!] = \{\ ({}^{\backprime}W{}^{\backprime}t, W't') \mid {}^{\backprime}W{}^{\backprime}t \models {}^{\backprime}\mathit{pre} \Rightarrow W't' \models \mathit{post}'\ \}$$

$$\mathcal{P}\rho[\![{}^{\backprime}\mathit{pre} \mid \mathit{dur} \mid \mathit{post}']\!] = \{\ ({}^{\backprime}W{}^{\backprime}t, Wt) \mid {}^{\backprime}W{}^{\backprime}t \models {}^{\backprime}\mathit{pre} \Rightarrow Wt \models \mathit{dur}\ \}$$

The logical notation used is first-order classical logic with modal operators of linear time. The latter abbreviate certain quantifications over the (scheduling) time variable (see Table II). The basic constructs are as follows: $\underline{\Box}\, p$, to be read "p holds now and is scheduled to hold forever in the future," $\underline{\Diamond} p$, to be read "p holds now or is scheduled to hold sometime in the future," and $\odot^{\tau} p$, to be read "p is scheduled to hold at a moment exactly τ units of time from now." Filled versions $\underline{\blacksquare}$, $\underline{\blacklozenge}$, $\bullet$, of these symbols refer backward in time. We use $p\, \underline{\textbf{until}}\, q$ for "p is scheduled to hold at least over an interval that stretches from now until the first moment at which q is scheduled to become true," $p\, \underline{\textbf{for}}\, \tau$ to mean "p is scheduled to hold throughout the coming τ units of time," and $p\, \underline{\textbf{after}}\, \tau$ to mean "p is scheduled to hold after τ units of time have elapsed." The values of time used for τ may be noninteger. It is always true that Wt models condition $\texttt{T} = t$ for the reserved variable $\texttt{T}$; this reads "in the tth state of a world line W, the time is t."

Informally, the specification $\rho[\grave{}pre \mid dur \mid post']$ is to be read as

> *if the specified process starts when pre holds, and it is inspected at some point during its subsequent execution, then dur will be found to hold then, and post will be found to hold when it terminates.*

So specifications denote processes with termination and suspension semantics lying in the same domains as the termination and suspension semantics of ordinary VHDL codes. The next section will give a meaning to *refinement* in this setting.

5. REFINEMENT

That the process `a` should implement the specification we have in mind for it will be written as

$$_Q[\grave{}T = 0 \mid Q = \lfloor T \,\mathbf{mod}\, 2 \rfloor \mid \mathit{false}] \sqsubseteq \mathtt{a}$$

where $\sqsubseteq$ is the *refinement relation.* This inequality means exactly that the termination and suspension denotations of `a` are subsets (subrelations) of the termination and suspension denotations of the specification. That is, if ρ is the list of signals controlled by the codes/specifications under consideration, then

$$b \sqsubseteq a \ \ \mathit{iff} \ \ \mathcal{S}\rho[\![b]\!] \supseteq \mathcal{S}\rho[\![a]\!] \ \mathit{and}\ \mathcal{P}\rho[\![b]\!] \supseteq \mathcal{P}\rho[\![a]\!].$$

Recall that specifications and code statements both lie in the domain of this relation. A specification may refine a specification.

This refinement relation is transitive, and all constructions of code and specifications in which we will be interested are monotonic (respect the order) with respect to it. That is, for codes or specifications a, b, c, and for any constructor $f[\_]$, such as, for instance, `while true do _`, we have the following:

$$a \sqsubseteq b \ \mathit{and}\ b \sqsubseteq c \ \mathit{implies}\ a \sqsubseteq c \qquad a \sqsubseteq b \ \mathit{implies}\ f[a] \sqsubseteq f[b]$$

When a constructor is applied to a less well determined specification then the result is less well determined.

The top element of the refinement ordering is $[\mathit{true} \mid \mathit{false} \mid \mathit{false}]$. This "miraculous" specification refines any specification. There is no code that implements it, but it is sometimes useful as a placeholder. The bottom element is $[\mathit{false} \mid \mathit{true} \mid \mathit{true}]$. Conjunction of specifications denotes their intersection ('$\wedge$') as relations. The following fact will be needed: if two specifications are refined by the same code, then their conjunction is also refined by the same code:

$$a \sqsubseteq c \ \mathit{and}\ b \sqsubseteq c \ \ \mathit{implies}\ \ a \wedge b \sqsubseteq c$$

The conjunction in the case when the preconditions are identical is obtained by the conjunction of the during- and postconditions, respectively:

$$\rho[\grave{}pre|dur_1|post'_1] \wedge \rho[\grave{}pre|dur_2|post'_2] = \rho[\grave{}pre|dur_1 \wedge dur_2|post'_1 \wedge post'_2]$$

No other instance than this will be required.

The development of the VHDL code `a` from the specification $_Q[\grave{}T = 0 \mid Q = \lfloor T \,\mathbf{mod}\, 2 \rfloor \mid \mathit{false}]$ will consist of a series of steps

$$_Q[\grave{}T = 0 \mid Q = \lfloor T \,\mathbf{mod}\, 2 \rfloor \mid \mathit{false}] = s_0 \sqsubseteq s_1 \sqsubseteq \ldots \sqsubseteq s_{n-1} \sqsubseteq s_n = \mathtt{a}$$

in which the stepping stones are hybrids, each consisting partly of a specification and partly of a VHDL description. Each refinement step will be formally justified by means of one of a set of *refinement laws*, introduced below.

6. REFINEMENT STEPS

We may suppose that the process `a` specified as follows in Section 5

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

is a single VHDL process with body `R`:

$$\mathtt{a} \;=\; \mathtt{process}\;[\mathtt{Q} : \mathtt{out}]\;\mathtt{begin}\;\mathtt{R}\;\mathtt{end}$$

This is the first design decision: to implement `a` as a monolithic process. The alternative would be to decompose the design into two cooperating subprocesses.

A single VHDL process is just a continuously running loop preceded by an initialization of the governed signals to "zero," the default initializing value for the type. We will take the default as 0. The initialization sets $\odot^{T} Q = 0$ for all scheduled values $\odot^{T} Q$ of the output `Q`, for all $T \geq 0$.

In the following, $\Box\,(Q = 0)$ denotes that $Q = 0$ is true at the present time and is scheduled to hold at all future times (precisely, $\forall t \geq 0 \bullet \odot^{t}(Q = 0)$). The specification that the implicit `while` loop in the process body has to satisfy is

$${}_{Q}[\,\Box\,(Q = 0) \wedge {}^{\backprime}T = 0 \mid Q = \lfloor T \,\mathbf{mod}\, 2 \rfloor \mid \mathit{false}] \sqsubseteq \mathtt{while}\;\mathtt{true}\;\mathtt{do}\;\mathtt{R} \tag{1}$$

It should be emphasized that the precondition asserts a *schedule* and a *current time*. These are statements about what is planned to happen. At time ${}^{\backprime}T = 0$ it is scheduled that Q will be zero from then on.

The above reduction of a VHDL process refinement to a simple continuous loop refinement is codified in Law 1 below. The horizontal rule separates the conclusion justified by the law (below the line) from the hypotheses necessary to its application (above the line; an instance is (1)):

LAW 1. *A specification $\rho[\,{}^{\backprime}pre \mid dur \mid \mathit{false}]$ may be refined to a single VHDL process if it may be refined to a continuous loop with the same body, plus an extra initialization of all the controlled signals to zero. Let pre imply no restrictions on ρ. Then*

$$\frac{\rho[\,\Box\,(\rho = 0) \wedge {}^{\backprime}pre \mid dur \mid post'] \sqsubseteq \mathtt{while}\;\mathit{true}\;\mathtt{do}\;R}{\rho[\,{}^{\backprime}pre \mid dur \mid post'] \sqsubseteq \mathtt{process}\;[\rho : \mathtt{out}]\;\mathtt{begin}\;R\;\mathtt{end}}.$$

The general case (when *pre* contains references to ρ) is more complicated but does not arise in practice. This is not a point that deserves dwelling on; an initial precondition for a single VHDL process should not constrain the output signals of the process because they are initialized by VHDL to some definite value as the process starts up. It would be pointless to constrain them differently.

Continuing to satisfy the subgoal (1) while refining the code part further is a procedure that will end with code satisfying the overall specification.

6.1 Loops

The problem of refining the loop in (1) can be reduced to the problem of refining the loop body. First design a condition I that the loop is to reestablish after every iteration. Then treat the *invariant* I as the pre- and postcondition seen by the loop body. The loop body has to set up the required during-condition:

$$ {}_Q[\,`I \wedge `T = t_n \mid Q = \lfloor T \,\mathbf{mod}\, 2 \rfloor \mid I' \wedge T' = t_{n+1}] \sqsubseteq \mathtt{R} \tag{2a}$$

Now, the invariant is to hold at the beginning and end of every pass through the loop. The nth cycle will begin at time t_n, the zeroth at time $t_0 = 0$. So first of all, the invariant must be in position after zero iterations at time $`T = t_0 = 0$, on entry into the loop:

$$\underline{\Box}\,(Q = 0) \wedge `T = 0 \rightarrow `I \tag{2b}$$

The general law here — covering also `while` loops with a test — is as follows. The initialization and termination constraints are wriiten as side-conditions:

LAW 2. *Let $t_0 \leq t_1 \leq \ldots$ be an increasing series. Then*

$$\frac{\rho[\,`I \wedge `x \wedge `T = t_n \mid dur \mid I' \wedge T' = t_{n+1}] \sqsubseteq a}{\rho[\,`pre \mid dur \mid post] \sqsubseteq \mathtt{while}\ x\ \mathtt{do}\ a} \begin{bmatrix} \forall \rho | pre \bullet T = t_0 \wedge I \\ \forall \rho | \neg x \wedge I \bullet post \end{bmatrix}.$$

Let us suppose now that each loop iteration takes one unit of time. This is another design decision — there are less regular ways than this of dividing up the work:

$$t_{n+1} - t_n = 1$$

Then the nth iteration begins at the nth instant of time, since $t_0 = 0$:

$$t_n = n$$

What is the loop invariant? If the design is successful, then at the beginning of the nth iteration it will be true that the output Q takes the value $n \,\mathbf{mod}\, 2$, since the time $T = n$ then and since $Q = \lfloor T \,\mathbf{mod}\, 2 \rfloor$ is the specification. At the end of that loop iteration, time has advanced to $T = n + 1$, and it will be true that the output Q takes the value $(n+1) \,\mathbf{mod}\, 2$. More will be true, that Q is scheduled to change in the next instant, for example, and that it will not change until then. In the context of continuous time it is necessary to stipulate that not only is $Q = \lfloor T \,\mathbf{mod}\, 2 \rfloor$ *now*, but that things are scheduled to stay that way during at least the next instant of time. In a discrete-time simulation, *now* implies *throughout the next unit instant of time* as well. The predicate $p \,\underline{\mathbf{for}}\, \tau$ denotes that p holds throughout a period of time commencing now and ending in (i.e., just before) τ units of time from now. That is, $\forall t | T \leq t < T + \tau \bullet p[t/T]$. A suitable continuous time invariant I is then

$$Q = \lfloor T \,\mathbf{mod}\, 2 \rfloor \,\underline{\mathbf{for}}\, 1.$$

Now (2b) is satisfied, because it is true that $\underline{\Box}\ Q = 0$ holds at time zero:

$$\underline{\Box}\,(Q = 0) \wedge T = 0 \rightarrow Q = \lfloor T \,\mathbf{mod}\, 2 \rfloor \,\underline{\mathbf{for}}\, 1$$

An invariant has now been designed, and it remains to design a loop body `R` that satisfies the subgoal (2a). This says that the loop body must reestablish the invariant on termination and that it must make use of the invariant initially in order to set up the during-condition. The next sections deal with that refinement.

6.2 Wait Statements

In order to satisfy (2a), we have to aim at letting the loop body R implement a flip of the output value in one unit of time. The following is (2a) written with the time t_n at which the nth loop iteration starts instantiated to n and with the loop invariant I instantiated to $Q = \lfloor T\,\mathbf{mod}\,2\rfloor\ \underline{\mathbf{for}}\ 1$:

$$_Q[Q{=}\lfloor T\,\mathbf{mod}\,2\rfloor\ \underline{\mathbf{for}}\ 1 \wedge {}^{\backprime}T{=}n \mid Q{=}\lfloor T\,\mathbf{mod}\,2\rfloor \mid Q{=}\lfloor T\,\mathbf{mod}\,2\rfloor\ \underline{\mathbf{for}}\ 1 \wedge T'{=}n{+}1] \\ \sqsubseteq \mathtt{R} \tag{3}$$

Suppose that the loop body R terminates in a `wait` statement, as is usual in VHDL process designs. This is another design decision:

$$\mathtt{R} = \mathtt{R}'; \mathtt{wait\ on\ Q}$$

Then it is sufficient, and necessary — because a `wait` can make no assignments — to assume that the postcondition $Q = \lfloor T\,\mathbf{mod}\,2\rfloor\ \underline{\mathbf{for}}\ 1 \wedge T' = n+1$ has been set up by R′ before entering the `wait`. What we will do in the following paragraphs is to use knowledge of the behavior of a VHDL `wait` statement in order to reduce the problem of finding R satisfying (3) to the problem of finding an R′ satisfying a certain other specification.

Now, the simplest design is that the `wait` should last throughout the one unit of time that R must execute over in this refinement example. The specification for R says that Q is to be constant for that one unit of time, and letting the `wait` statement take up all that interval will ensure that it is so (this process controls Q; if it is busy waiting then it cannot be writing, and Q will remain constant). But the `wait on Q` must terminate after the one unit of time because R terminates then. A `wait on Q` terminates only when the schedule says to change the value of Q, so $Q \neq\ \odot Q$ must be part of the precondition for the `wait`. But we still need to say more — namely, that Q is constant until the change in one unit's time and that thereafter it is constant for one unit of time (satisfying the postcondition). Let $p\,\underline{\mathbf{then}}\,q$ be the temporal expression saying that the interval of time during which p is true terminates at a moment when q is true (precisely, $p\,\underline{\mathbf{until}}(q \wedge \neg p)$), so we need the following precondition to the `wait`:

$$Q = \lfloor T\,\mathbf{mod}\,2\rfloor\ \underline{\mathbf{for}}\ 1 \quad \underline{\mathbf{then}} \quad Q = \lfloor T\,\mathbf{mod}\,2\rfloor\ \underline{\mathbf{for}}\ 1$$

This reduces to the more succinct

$$Q = \lfloor T\,\mathbf{mod}\,2\rfloor\ \underline{\mathbf{for}}\ 2$$

which is a strong enough precondition for the `wait` to force the required during- and postcondition. Formally, the refinement being suggested is

$$_Q[Q{=}\lfloor T\,\mathbf{mod}\,2\rfloor\ \underline{\mathbf{for}}\ 2 \wedge {}^{\backprime}T{=}n \mid Q{=}\lfloor T\,\mathbf{mod}\,2\rfloor \mid Q{=}\lfloor T\,\mathbf{mod}\,2\rfloor\ \underline{\mathbf{for}}\ 1 \wedge T'{=}n{+}1] \\ \sqsubseteq \mathtt{wait\ on\ Q} \tag{4}$$

This is the conclusion of Law 3 below. Let $p\,\underline{\mathbf{while}}\,q$ be the temporal logic statement which says that p endures at least as long as condition q (precisely, $p\,\underline{\mathbf{until}}\,\neg q$):

Law 3. *If pre forces the first change in Q to occur at a time when post holds, then the specification triple can be implemented by a* `wait on` *Q statement:*

$$\frac{}{\rho[{}^{\backprime}pre \mid dur \mid post'] \sqsubseteq \mathtt{wait\ on}\ Q}$$

provided that

$$\forall \rho | pre \bullet \exists q | Q{=}q\, \underline{\mathbf{then}}\, post \wedge dur\, \underline{\mathbf{while}}\, Q{=}q$$

The side-conditions to the law read "Q is constant for a while and then *post* holds, and *dur* holds while Q is constant."

We are on the way to satisfying the loop body refinement (3). The postcondition and during-condition are set up correctly in the final `wait` statement, and we need to ensure only that the during-condition persists in the preceding statements $\mathtt{R}'$ too. The next sections deal with this requirement.

6.3 Sequence

Having determined an appropriate precondition for the `wait`, the requirements on $\mathtt{R}'$ are forced. The precondition for the `wait` is the postcondition for the statement that precedes it. The during-condition for the loop must hold not only through the `wait` statement, but during the preceding statements too. The precondition for the loop body is given. That is

$$\begin{aligned}&{}_Q[Q{=}\lfloor T\,\mathbf{mod}\,2\rfloor\ \underline{\mathbf{for}}\ 1 \wedge {}^{\backprime}T{=}n \mid Q{=}\lfloor T\,\mathbf{mod}\,2\rfloor \mid Q{=}\lfloor T\,\mathbf{mod}\,2\rfloor\ \underline{\mathbf{for}}\ 2 \wedge T'{=}n]\\ &\sqsubseteq \mathtt{R}' \qquad (5)\end{aligned}$$

This reasoning is codified in the following law:

Law 4. *A specification may be refined to a sequence of statements if the preconditions and postconditions of the parts of the sequence match up together, and if the during-condition holds throughout.*

$$\frac{\rho[{}^{\backprime}pre \mid dur \mid mid'] \sqsubseteq R_1 \qquad \rho[{}^{\backprime}mid \mid dur \mid post'] \sqsubseteq R_2}{\rho[{}^{\backprime}pre \mid dur \mid post'] \sqsubseteq R_1; R_2}$$

In the present case, the law is applied with `wait on Q` as R_2 and $\mathtt{R}'$ as R_1, leaving as subgoals (4), which has already been shown to hold, and (5).

6.4 Transport Assignment

The specification (5) for $\mathtt{R}'$ can be achieved by letting it be a single (transport) assignment statement:

$$\mathtt{R}' = Q \Leftarrow \neg Q\ \mathtt{after}\ 1$$

This would satisfy the requirement (5), according to Law 5 below. In it, a new *logical constant* t is introduced to capture an initial time value T and save it for later comparison. t may not appear in *pre* or *post*. Similarly, x captures the initial value of expression X. The term "$T \geq t{+}\tau?x : Q$" is borrowed from the C language and should be read "if the current time $T \geq t + \tau$, then x, else Q." The formula $post[(T \geq t + \tau?x : Q)/Q]$ is *post* with every occurrence of Q in it replaced by the above-mentioned term.

Law 5. *A transport assignment of expression X to Q after τ satisfies a specification if the precondition implies the postcondition when the current value x of X replaces the appearance of any scheduled value of Q from τ onward:*

$$\overline{\rho[{}^{\backprime}pre \mid dur \mid post'] \sqsubseteq Q \Leftarrow X\ \texttt{after}\ \tau}$$

provided that

$$\forall \rho \mid {}^{\backprime}pre \bullet \exists t = {}^{\backprime}T, x = X \mid post'[(T' \geq t + \tau ? x : Q)/Q].$$

The during-condition is not of interest here. A transport assignment cannot be inspected during its execution, and that is what the unconstrained *dur* in the law means. Anything at all is allowed to be true in the nonexistent interval of time in which the transport assignment executes. Executing a transport assignment does not imply anything for during-conditions.

Here *post* is $T = n \wedge Q = \lfloor T \,\mathbf{mod}\, 2 \rfloor \,\underline{\mathbf{for}}\, 2$. That is,

$$T = n \wedge \lfloor Q = \lfloor T \,\mathbf{mod}\, 2 \rfloor \,\underline{\mathbf{for}}\, 1 \,\underline{\mathbf{then}}\, Q = \lfloor T \,\mathbf{mod}\, 2 \rfloor \,\underline{\mathbf{for}}\, 1$$

The law says that the precondition must imply *post* in which any occurrences of Q further than or equal to $\tau = 1$ in the future have been replaced by $\neg q$, the negation of the present value of Q, giving

$$T = n \wedge Q = n \,\mathbf{mod}\, 2 \,\underline{\mathbf{for}}\, 1 \,\underline{\mathbf{then}}\, \neg(n \,\mathbf{mod}\, 2) = (n+1) \,\mathbf{mod}\, 2 \,\underline{\mathbf{for}}\, 1.$$

And this is indeed implied by *pre*, which is $T = n \wedge Q = T \,\mathbf{mod}\, 2 \,\underline{\mathbf{for}}\, 1$.

The body of the loop is now a transport assignment followed by a `wait` statement.

$$R \;=\; \texttt{Q} \Leftarrow \neg\texttt{Q after 1; wait on Q}$$

That is the end of the derivation. The initial specification is satisfied by the following design:

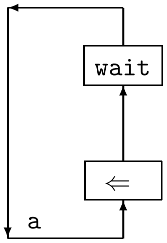
```
a: process [Q:out] begin
  Q ⇐ ¬ Q after 1 ;
  wait on Q ;
end
```

which oscillates with half-period 1. A control flow diagram for the derived code is shown in Figure 4. The derivation has been rather longer than it needed to be for the purposes of introducing and justifying the logical laws involved.

7. FURTHER EXAMPLE — AN ASYNCHRONOUS RS FLIP-FLOP

In this section a slightly more substantial piece of hardware design is considered: a single-element *RS flip-flop.* This is a unit from which more complicated designs may easily be constructed, and it is important to get the basic design right.

The single element RS flip-flop is a unit with two boolean inputs S and R (“set” and “reset,” respectively) and two boolean outputs Q and $\overline{Q}$. The latter are usually but not always complements. If Q is *true* (and $\overline{Q}$ is *false*), then the flip-flop is said to be “full” or “set.” If Q is *false* (and $\overline{Q}$ is *true*), then the flip-flop is said to be “empty” or “unset.” An unset flip-flop can be set and then will become set, after a



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

Fig. 4. Process "a" VHDL description and control flow diagram.

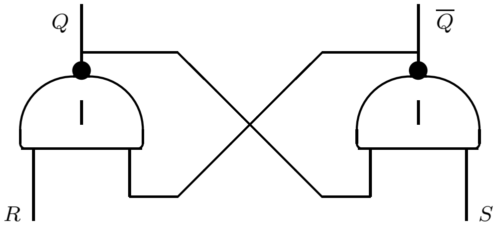


Fig. 5. A standard RS flip-flop circuit contains two NOR-gates wired together.

short delay. A set flip-flop can be reset and will become unset (again, after a short delay). We will suppose that the delay is always a single unit of time.

It is not straightforward to write a continuous-time specification that is realistic but not over-constraining. The general experience is of circuits in which a system clock discretizes the passage of time and what the RS flip-flop should do between clock pulses, as opposed to what a particular circuit does do, is not well established. It is not clear, for example, quite how long the flip-flop needs to hold its outputs stable, or when it needs to start doing so, or what it should do while changing output values, or how long the inputs need to be stable, and when.

Consider first the usual discrete-time implementation shown in Figure 5. It consists of a pair of NOR-gates in a feedback configuration (sometimes also with a pair of NAND-gates guarding the inputs). With a delay of one unit of time between input and output, the following pair of equations defines the output. Recall that the filled-in circle denotes "one unit of time ago." This is necessarily a backward-oriented specification because forward references in time refer only to a VHDL driver set (schedule), not to reality:

$$\begin{aligned}\neg Q &= \bullet\ (R \vee \overline{Q})\\ \neg\overline{Q} &= \bullet\ (S \vee Q)\end{aligned}$$

In any time domain, these equations have least solution:

$$\begin{array}{l}\neg Q = \bullet\, R \;\vee\; \bullet^{2}\neg S \wedge \bullet^{3} R \;\vee\; \bullet^{2}\neg S \wedge \bullet^{4}\neg S \wedge \bullet^{5} R \;\vee\; \ldots \\ \neg \overline{Q} = \bullet\, S \;\vee\; \bullet^{2}\neg R \wedge \bullet^{3} S \;\vee\; \bullet^{2}\neg R \wedge \bullet^{4}\neg R \wedge \bullet^{5} S \;\vee\; \ldots\end{array}$$

Surprisingly, a reset applied two units of time ago and not subsequently countermanded by a set command will not reset this flip-flop (put it in a state with $\neg Q$). It will do so one unit of time in the future. Yet a reset applied one or three units of time ago will reset the flip-flop now. A reset applied any even number of units of time ago will have no effect until a unit of time in the future, while a reset applied an odd number of units of time ago will have an effect now. It is necessary to stipulate that *each set or reset command lasts at least through two clock points.*

Under these circumstances, the flip-flop obeys a simple specification that only needs a temporal **since** operator for its proper expression. $a\,\underline{\textbf{since}}\,b$ is the time-reversed version of $a\,\underline{\textbf{until}}\,b$. It means that there is some time $t < 0$ (i.e., in the past) for which $\odot^{t} b$ holds, and for all τ in the ensuing time interval $(t, 0]$ $\odot^{\tau} a$ holds, or else a has always held. The following is the specification for the flip-flop:

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

In natural language, the current state of the flip-flop depends on the inputs in the past. If the flip-flop is unset now, then it is because until "recently" (i.e., one unit of time ago) the last event to be registered had not been a set signal.

Recall that it is assumed that *all signals endure at least through some two clock points one unit of time apart.* Under these conditions the formal specification for the RS flip-flop can be as follows:

$${}_{Q,\overline{Q}}\left[T = 0 \,\middle|\, \begin{array}{lcl}\neg Q &\leftrightarrow& \bullet\,(\neg S\,\underline{\textbf{since}}\,R) \\ \neg\overline{Q} &\leftrightarrow& \bullet\,(\neg R\,\underline{\textbf{since}}\,S)\end{array} \,\middle|\, true\right]$$

Now, a specification can always be strengthened, and any refinement of the strengthened version will satisfy the original. That is codified in the following law:

Law 6. *A precondition can be strengthened, or a during- or postcondition can be weakened without affecting the validity of a refinement step.*

$$\frac{\rho[pre \mid dur \mid post] \sqsubseteq a}{\rho[pre' \mid dur' \mid post'] \sqsubseteq a}[\forall\rho|pre' \bullet pre \wedge \forall\rho|dur \bullet dur' \wedge \forall\rho|post \bullet post']$$

The law is never absolutely necessary to a derivation, because its use can be avoided by repeating some calculations with minor modifications, but it is usually very convenient in practice. In the present case we can apply it to replace the specification with the recursive form that corresponds to the classical RS flip-flop design, because the latter satisfies the specification:

$${}_{Q,\overline{Q}}\left[T = 0 \,\middle|\, \begin{array}{lcl}\neg Q &\leftrightarrow& \bullet\,(R \vee \overline{Q}) \\ \neg\overline{Q} &\leftrightarrow& \bullet\,(S \vee Q)\end{array} \,\middle|\, true\right]$$

But while the equations solve in the discrete-time setting under certain assumptions on signal longevity and separation, there is no a priori guarantee that they solve in general to maintain those assumptions. The subject is treated again a few paragraphs lower down.

The next step in the refinement is to decompose the specification into two parts `r` and `l` that can be refined separately by synchronously acting processes. One process (`r`) will control the Q line, and the other (`l`) will control the $\overline{Q}$ line. They will both see each others output as well as the S and R lines. The during-conditions for the two processes are respectively (those for `l` on the left and for `r` on the right):

$$\neg\overline{Q} \leftrightarrow \bullet\ (S \vee Q) \,\|\, \neg Q \leftrightarrow \bullet\ (R \vee \overline{Q}) \tag{6}$$

Both subprocesses are acting in concert to maintain Q and $\overline{Q}$ in step with each other. This reduction to two subprocesses is formally codified in the law below:

Law 7. *Two derivations which rely on the during-conditions obtained in the other may be combined into one derivation of a parallel pair, eliminating the hypotheses.*

$$\dfrac{\begin{array}{c}[\forall\rho_2 / \bullet\ dur] \\ \vdots \\ {}_{\rho_1}[pre \mid dur \mid post] \sqsubseteq R_1\end{array} \qquad \begin{array}{c}[\forall\rho_1 / \bullet\ dur] \\ \vdots \\ {}_{\rho_2}[pre \mid dur \mid post] \sqsubseteq R_2\end{array}}{{}_{\rho_1\rho_2}[pre \ \mid \ dur \ \mid \ post] \sqsubseteq R_1 \,||\, R_2}$$

In the following, we will consider the refinement of subprocess `r`. The refinement for `l` is symmetrical. Suppose that `r` is a monolithic process with body `r′`:

$$\texttt{r} \ = \ \texttt{process}\,[\texttt{Q} : \texttt{out}]\ \texttt{begin}\ \texttt{r}'\ \texttt{end}$$

Then, as before, it can be assumed that Q is initialized to zero by the process header and then that an infinite loop with body `r′` is entered. The specification is

$${}_{Q}[T = 0 \wedge \underline{\Box}\ \neg Q \mid \neg Q \leftrightarrow \bullet\ (R \vee \overline{Q}) \mid true] \sqsubseteq \texttt{while}\ true\ \texttt{do}\ \texttt{r}'.$$

An invariant I for the loop body is needed that will force the during-condition. Call the during condition $\neg Q \leftrightarrow \bullet\ (R \vee \overline{Q})$ in the above dur. An invariant is

$$dur\ \underline{\textbf{for}}\ 1$$

This states that, over the next unit of time, Q is scheduled to follow the changes that are expected of it. At the end of that period of time, Q is scheduled to take the value that $\neg(R \vee \overline{Q})$ has now, and at the beginning of that period of time it takes the value that $\neg(R \vee \overline{Q})$ had a unit of time ago.

When the loop is entered for the first time, the invariant may be considered already to be in place. There are at that point no signals one unit of time ago, since the time referred to is before the start point of the process; but for convenience we stipulate that the reset line has been high up till the zero point:

$$\underline{\blacksquare}\ R$$

That allows the base case for the loop invariant to be satisfied. Now, let the nth iteration of the loop start at t_n. The iterative case for the loop invariant is then

$${}_{Q}[T = t_n \wedge dur\ \underline{\textbf{for}}\ 1 \mid dur \mid T = t_{n+1} \wedge dur\ \underline{\textbf{for}}\ 1] \sqsubseteq \texttt{r}'.$$

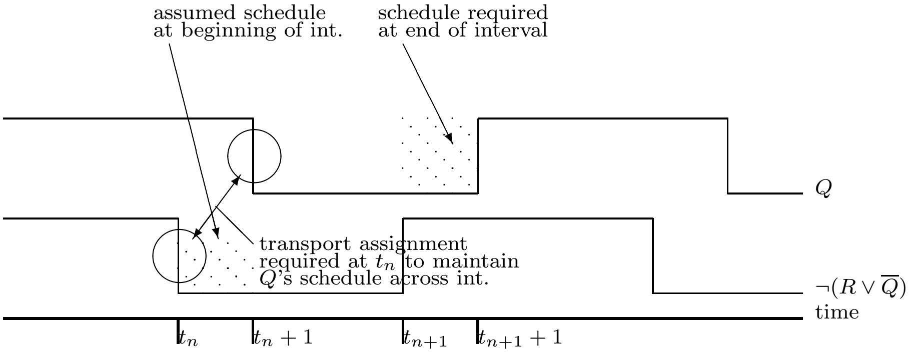

assumed schedule at beginning of int.
schedule required at end of interval
transport assignment required at t_n to maintain Q's schedule across int.
Q
$\neg(R \vee \overline{Q})$
time
t_n
t_n+1
t_{n+1}
$t_{n+1}+1$

Fig. 6. The schedule $dur\,\underline{\mathbf{for}}\,1$ at the beginning of the interval $[t_n, t_{n+1})$, where dur denotes that Q follows $\neg(R\vee\overline{Q})$ lagging by one unit of time, is set up for the end of the interval via a transport assignment $Q \Leftarrow \neg(R \vee \overline{Q})$ `after` 1 at t_n. R and $\overline{Q}$ are constant during the interval, which is much longer than one unit of time in length.

Following the usual VHDL convention, the loop body will end with a `wait` statement:

$$\mathtt{r}' = \mathtt{r}'' \,;\mathtt{wait\ on}\ \overline{\mathtt{Q}}, \mathtt{R}$$

It is then simplest to design the intervals $[t_n, t_{n+1})$ to be those in which none of R and $\overline{Q}$ change, but where there is a change in one of R or $\overline{Q}$ at each t_n (except $t_0 = 0$). Furthermore, we suppose that *each of these intervals is at least one unit of time long.* That is, changes in R and $\overline{Q}$ do not occur within one unit of time of the last change to either. At this point, moreover, we return to the problem mentioned a few paragraphs higher up — namely, that we cannot be sure that this assumption on signal behavior is maintained through the mutual recursion that is being solved. Our solution is to add as a global restriction that *no signal can change value at points separated by real time of less than one unit duration.*

This assumption properly reflects the behavior of the real components when VHDL is implemented. It also allows changes to take place in infinitesimal (not real time) intervals, as in VHDL simulations.

Now, for simplicity, we let the `wait` statement begin execution at the beginning of the interval $[t_n, t_{n+1})$ and occupy the whole interval, exiting at the end of the interval, when there is a change in one of R or $\overline{Q}$. A precondition for the `wait` that will force $dur\,\underline{\mathbf{for}}\,1$ at the end of this interval (which lasts $t_{n+1} - t_n$ units of time) is $dur\,\underline{\mathbf{for}}\,t_{n+1} - t_n + 1$, since there are no events during the `wait`. The following is then a specification for $\mathtt{r}''$:

$${}_{\mathtt{Q}}[\mathtt{T} = \mathtt{t_n} \wedge \mathtt{dur}\,\underline{\mathbf{for}}\,\mathtt{1} \mid \mathtt{dur} \mid \mathtt{T} = \mathtt{t_n} \wedge \mathtt{dur}\,\underline{\mathbf{for}}\,\mathtt{t_{n+1}} - \mathtt{t_n} + \mathtt{1}] \sqsubseteq \mathtt{r}''$$

This specification is satisfied by a transport assignment of the current value of $S \wedge \neg R$ to Q with a delay of 1. The illustration in Figure 6 will help to see why.

In the period $[1, 1 + t_{n+1} - t_n)$ no more changes will happen, and Q will continue to follow the (unchanging) value of $\neg(R \vee \overline{Q})$. And a transport assignment takes no time, so the during-condition is satisfied vacuously, i.e.,

$$\mathtt{r}'' \;=\; \mathtt{Q}\Leftarrow \neg\mathtt{R} \wedge \neg\overline{\mathtt{Q}}\ \mathtt{after\ 1}$$

So the complete code for `r` is

```
process [Q:out]
begin
  Q ⇐ not R and not Q̄ after 1 ;
  wait on R, Q̄
end
```

The code for `l` is the mirror image. `S` replaces `R`, and $\overline{\mathtt{Q}}$ replaces `Q`.

That concludes the presentation of the refinement rules for VHDL. In the next section their correctness and applicability will be examined.

8. A FORMAL LOGIC

In this section the background required to state and prove completeness of the refinement calculus will be sketched. The approach is via formal logic. An axiomatic logic of equal proof-theoretic power to the refinement calculus will be presented.

8.1 Generalities

A classical approach to programming languages is to regard a program or program statement as being modeled by a *nondeterministic change of state*:

$$\mathcal{S}[\![code]\!] : State_{Init} \leftrightarrow State_{Final}$$

This semantics induces a representation in another domain. A set p of possible initial states is related to a set of possible final states q if every possible run from an initial state ${}^{\backprime}s \in p$ ends with a final state $s' \in q$. This is the *Hoare program semantics* (here $(\!|x|\!)R$ denotes the image set of x under R):

$$\begin{array}{l} \mathcal{H}[\![code]\!] : \mathrm{P}\, State_{Init} \leftrightarrow \mathrm{P}\, State_{Final} \\ (p,q) \in \mathcal{H}[\![code]\!] \textit{ iff } (\!|p|\!)\mathcal{S}[\![code]\!] \subseteq q \end{array}$$

The Hoare semantics can regenerate the original change of state:

$$({}^{\backprime}s, s') \in \mathcal{S}[\![code]\!] \textit{ iff } s' \in \bigcap_{\{{}^{\backprime}s\}\mathcal{H}[\![code]\!]q} q$$

So the two representations are isomorphic if only (nondeterministic) changes of state are being represented. Hoare semantics may represent more, however.

There is a third semantics. Any Hoare semantics gives rise to a *weakest precondition semantics* and vice versa. Given a code *code* and a particular set of final states q, there is a maximal set of initial states p such that any state ${}^{\backprime}s \in p$ is bound to run through *code* to a state $s' \in q$. p is called the *weakest precondition* of q:

$$wp(code, q) = \bigcup_{p\mathcal{H}[\![code]\!]q} p$$

The Hoare semantics generated from a weakest precondition semantics is

$$p\mathcal{H}[\![code]\!]q \quad \textit{iff} \quad p \subseteq wp(code, q).$$

And when the Hoare semantics is derived from a change of state semantics, then the weakest precondition semantics regenerates the Hoare semantics by this mechanism. So all three views are equivalent on nondeterministic state machines.

Refinement calculi are closely related to Hoare semantics. To a pre-/postcondition specification $[p \mid q]$ is given the semantics of the relation between states $`s$, $s'$ such that if the initial state $`s$ satisfies p, then the final state s' satisfies q:

$$\mathcal{S}[\![p \mid q]\!] = \{\ (`s, s') \mid p\ `s \rightarrow q\ s'\ \}$$

So

$$\mathcal{S}[\![p \mid q]\!] \supseteq \mathcal{S}[\![code]\!]\ \mathit{iff}\ (\!|p|\!)\mathcal{S}[\![code]\!] \subseteq q\ \mathit{iff}\ p\mathcal{H}[\![code]\!]q$$

by the definition of Hoare semantics above. This is the *refinement relation*:

$$[p \mid q] \sqsubseteq code\ \ \mathit{iff}\ \mathcal{S}[\![p \mid q]\!] \supseteq \mathcal{S}[\![code]\!]$$

It expresses the idea that the code does what the specification says it does.

8.2 Specifics

We associate two Hoare logics for VHDL with the refinement calculus set out in Sections 6 and 7. Instead of writing that process a refines the specification $_Q[`T = 0 \,|\, Q = \lfloor T\,\mathbf{mod}\,2\rfloor \,|\, \mathit{false}]$, we will say that if $T = 0$ holds when process a starts, then $Q = \lfloor T\,\mathbf{mod}\,2\rfloor$ will hold when it is suspended, and that if $T = 0$ holds when process a starts, then *false* will hold when it terminates:

$$\mathcal{P}_Q : T = 0\ \{\mathtt{a}\}\ Q = \lfloor T\,\mathbf{mod}\,2\rfloor\ \ \mathit{and}\ \ \mathcal{S}_Q : T = 0\ \{\mathtt{a}\}\ \mathit{false}$$

The two logics $\mathcal{S}$ and $\mathcal{P}$ are associated with the denotational semantics of termination and suspension respectively.

We give the deduction rules of the systems $\mathcal{S}$ and $\mathcal{P}$ first. The refinement calculus laws are derived from these systems via the following syntactic correspondence:

$$\rho[`pre \,|\, dur \,|\, true] \sqsubseteq a\ \leftrightarrow\ \mathcal{P}_\rho : pre\ \{a\}\ dur$$

$$\rho[`pre \,|\, true \,|\, post'] \sqsubseteq a\ \leftrightarrow\ \mathcal{S}_\rho : pre\ \{a\}\ post$$

The correspondence is also semantically sound. Conversely, the logical systems $\mathcal{S}$ and $\mathcal{P}$ can be obtained from the laws of the refinement calculus from the correspondence above and the following fact about specifications:

$$\rho[`pre|dur|true] \sqsubseteq a\ \mathit{and}\ \rho[`pre|true|post'] \sqsubseteq a\ \mathit{iff}\ \rho[`pre|dur|post'] \sqsubseteq a$$

From the proof-theoretic point of view, the systems are equivalent, as will be formalized below.

The set of deduction rules for the pre-/postcondition logic $\mathcal{S}\rho$ is shown in Figure 9, and the rules for the pre-/during-condition logic $\mathcal{P}\rho$ are shown in Figure 10. In the terminology, these are *weak* logics of termination and continuation respectively, in the sense that the asserted condition holds only *if* the statement terminates, or *while* the statement continues executing, as appropriate. So, since it is impossible that a `null` statement should execute over some nonzero duration, for example, we have the axiom that a `null` statement refines any continuation specification, or equivalently, satisfies any pre-/during-condition pair:

$$\frac{}{\rho[`pre \,|\, dur \,|\, true] \sqsubseteq \mathtt{null}}\quad\leftrightarrow\quad\frac{}{\mathcal{P}\rho : pre\ \{\mathtt{null}\}\ dur}$$

```
T=t ∧ ∃ q • □ Q=q
  { Q <= transport (not Q) after 1 }
T=t ∧ ∃ q • Q=q for 1 then □ Q=¬q
  { wait on Q }
T=t+1 ∧ ∃ q′ • Q=¬q′ for -1 then □ Q=q′
```

Reasoning with the logic of execution until termination.

```
T=n ∧ ∃ q′ • Q=¬q′ for -1 then □ Q=q′
  { Q <= transport (not Q) after 1 }
T=n ∧ ∃ q′ • Q=¬q′ for -1 then Q=q′ for 1 then □ Q=¬q′
  { wait on Q }
n≤T<n+1 ∧ ∃ q′ • Q=q′ for T-⌊T⌋ then Q=q′ for T-⌈T⌉ then □ Q=¬q′
```

Reasoning with the logic of execution until continuation.

Fig. 7. Proving via Hoare logic that the during-condition expressing the fact that signal Q has been stable and is about to change holds true of the code body of process a of Figure 4.

Reasoning in these logics is illustrated in Figure 7, which shows a verification that the derived code of Figure 4 really does oscillate.

We formally state the connection between the logic and the refinement calculus:

PROPOSITION 8.2.1. *$\rho[\grave{}pre \,|\, dur \,|\, post'] \sqsubseteq a$ is derivable in the refinement calculus iff it is provable in the logics that $\mathcal{P}\rho : pre \ \{a\}\ dur$ and $\mathcal{S}\rho : pre\ \{a\}\ post$.*

The proposition is proved by the following lemma plus the extra fact that the conjunction of two specifications with the same precondition is obtained by taking the conjunctions of the during- and postconditions:

LEMMA 8.2.2. *$\rho[\grave{}pre \,|\, dur \,|\, true] \sqsubseteq a$ is derivable in the refinement calculus iff it is provable in the logic that $\mathcal{P}\rho : pre\ \{a\}\ dur$, and $\rho[\grave{}pre \,|\, true \,|\, post'] \sqsubseteq a$ is derivable in the refinement calculus iff it is provable in the logic that $\mathcal{S}\rho : pre\ \{a\}\ post$.*

The proof is by structural induction on a and the proof of a. A refinement derivation may be rewritten line by line to look like a verification proof, and vice versa.

For example, suppose that we have proved $\mathcal{P}\rho : pre\ \{a;b\}\ dur$, in a proof terminating with the deduction:

$$\frac{\mathcal{P}\rho : pre\ \{a\}\ dur \quad \mathcal{S}\rho : pre\ \{a\}\ mid \quad \mathcal{P}\rho : mid\ \{b\}\ dur}{\mathcal{P}\rho : pre\ \{a;b\}\ dur}$$

Then, by induction, we may assume that we have proofs of the equivalents of the hypotheses in the refinement calculus, namely

$$\frac{\vdots}{\rho[pre \,|\, dur \,|\, true] \sqsubseteq a} \quad \frac{\vdots}{\rho[pre \,|\, true \,|\, mid] \sqsubseteq a} \quad \frac{\vdots}{\rho[mid \,|\, dur \,|\, true] \sqsubseteq b}.$$

$$wp(a;b,dur,post) = wp(a,dur,wp(b,dur,post)) \tag{7}$$

$$\begin{aligned} wp(\mathtt{if}\ x\ \mathtt{then}\ a\ \mathtt{else}\ b,dur,post) = {} & \mathtt{if}\ x\ \mathtt{then}\ wp(a,dur,post) \\ & \mathtt{else}\ wp(b,dur,post) \end{aligned} \tag{8}$$

$$wp(\mathtt{wait\ on}\ X,dur,post) = \exists x.(dur\ \underline{\mathbf{while}}\ x = X) \wedge (x = X\ \underline{\mathbf{then}}\ post) \tag{9}$$

$$wp(X \Leftarrow Y\ \mathtt{after}\ \tau,dur,post) = \exists t = T, y = Y | post[(T \geq t + \tau ? y : X)/X] \tag{10}$$

$$wp(\mathtt{null},dur,post) = post \tag{11}$$

$$wp(a \,||\, b,dur,post) = wp(a,dur,post) \wedge wp(b,dur,post) \tag{12}$$

$$wp([pre' \,|\, dur' \,|\, post'],dur,post) = pre' \rightarrow ((dur' \rightarrow dur) \wedge (post' \rightarrow post)) \tag{13}$$

$$\begin{aligned} wp(\mathtt{process}\ \sigma\ \mathtt{begin}\ R\ \mathtt{end},dur,post) = {} & \neg \underline{\Box}\,(\sigma{=}0) \\ & \vee wp(\mathtt{while}\ true\ \mathtt{do}\ R,dur,post) \end{aligned} \tag{14}$$

$$\begin{aligned} wp(\mathtt{while}\ x\ \mathtt{do}\ R,dur,post) = {} & \bigwedge_{n \geq 1} F^n[post] \\ & where\ F[p] = \neg x \wedge p \vee x \wedge wp(R,dur,p) \end{aligned} \tag{15}$$

Fig. 8. The combined weakest precondition semantics of continuation and termination. The infinitary form given for loops has the same denotation as a certain computable formula (see text).

The first two of these are inequalities on a, and we can take the conjunction, getting $\rho[pre \,|\, dur \,|\, mid] \sqsubseteq a$. Together with the third inequality and the law of sequence refinement (Law 4) we conclude that $\rho[pre \,|\, dur \,|\, true] \sqsubseteq a;b$, as required.

The question "if a code really satisfies a specification, then can it be proved that it does" now reduces to "is the logic *complete* with respect to the denotational semantics of VHDL" (listed formally in Table I). A logic is complete if anything that is expressible in the logic and which is a universal truth in the semantics may be proved in the logic. Conversely, the question of whether the refinement laws are correct reduces to the question of whether the logic is *sound* with respect to that denotation. A logic is sound if its rules are universally valid in the semantic domain.

PROPOSITION 8.2.3. *The refinement calculus is complete iff the logic is complete. The refinement calculus is sound iff the logic is sound.*

It is the case that the logic is both sound and complete. This was first proved in Breuer et al. [1995] for a weaker form of this logic. Soundness is a technical result requiring no more than checking that the semantics given supports each rule of the logic. Completeness is harder, but the proof will be sketched in the following section.

PROPOSITION 8.2.4. *The logic given here is sound.*

THEOREM 8.2.5. *The logic given here is complete.*

$$\frac{}{\mathcal{S}\rho : \mathit{pre}\ \{\texttt{null}\}\ \mathit{post}}[\forall\rho|\mathit{pre} \bullet \mathit{post}] \tag{16}$$

$$\frac{}{\mathcal{S}\rho : \mathit{pre}\ \{\texttt{wait on}\ X\}\ \mathit{post}}[\forall\rho|\mathit{pre} \bullet \exists x|X{=}x\, \underline{\mathbf{then}}\, \mathit{post}] \tag{17}$$

$$\frac{\mathcal{S}\rho : \mathit{pre}\ \{a\}\ \mathit{mid} \quad \mathcal{S}\rho : \mathit{mid}\ \{b\}\ \mathit{post}}{\mathcal{S}\rho : \mathit{pre}\ \{a\ ;\ b\}\ \mathit{post}} \tag{18}$$

$$\frac{}{\mathcal{S}\rho : \mathit{pre}\ \{X{\Leftarrow}\, Y\ \texttt{after}\ \tau\}\ \mathit{post}}[\forall\rho|\mathit{pre} \bullet \exists t{=}T, y{=}Y|\mathit{post}[(T{\geq}t{+}\tau?y{:}X)/X]] \tag{19}$$

$$\frac{\mathcal{S}\rho : \mathit{pre} {}_\wedge\, x\ \{b_1\}\ \mathit{post} \quad \mathcal{S}\rho : \mathit{pre} {}_\wedge \neg x\ \{b_0\}\ \mathit{post}}{\mathcal{S}\rho : \mathit{pre}\ \{\texttt{if}\ x\ \texttt{then}\ b_1\ \texttt{else}\ b_0\}\ \mathit{post}} \tag{20}$$

$$\frac{\mathcal{S}\rho : I {}_\wedge\, x\ \{b\}\ I \quad \mathcal{S}\rho : I {}_\wedge \neg x\ \{\texttt{null}\}\ \mathit{post}}{\mathcal{S}\rho : \mathit{pre}\ \{\texttt{while}\ x\ \texttt{do}\ b\}\ \mathit{post}}[\forall\rho|\mathit{pre} {}_\wedge\, x \bullet I] \tag{21}$$

$$\frac{\mathcal{S}\rho : \mathit{pre}\ \{a\}\ \mathit{post} \qquad \mathcal{S}\rho : \mathit{pre}\ \{b\}\ \mathit{post}}{\mathcal{S}\rho : \mathit{pre}\ \{a\,||\,b\}\ \mathit{post}} \tag{22}$$

$$\frac{}{\mathcal{S}\rho : \mathit{pre}\ \{[\mathit{pre}'\,|\,\mathit{post}']\}\ \mathit{post}}[\forall\rho|\mathit{pre} \bullet \mathit{pre}' {}_\wedge \forall\rho|\mathit{post}' \bullet \mathit{post}] \tag{23}$$

$$\frac{}{\mathcal{S}\rho : \mathit{pre}\ \{\texttt{process}\ \sigma\ \texttt{begin}\ R\ \texttt{end}\}\ \mathit{post}} \tag{24}$$

Fig. 9. Hoare logic for terminating execution of VHDL statements.

Corollary 8.2.6. *The refinement calculus is sound and complete.*

9. COMPLETENESS

Some intermediate results are required in order to prove the completeness theorem stated above. First, we extend the *weakest precondition* of a statement R to take account of both a during-condition *dur* and a postcondition *post*:

$$\rho : \mathit{wp}[R, \mathit{dur}, \mathit{post}]$$

is a precondition for R with the property that it forces the during-condition to hold during execution *and* forces the postcondition to hold after execution and is the mildest precondition that will do so.

The weakest preconditions for single during- and postconditions are as given below, and the double weakest precondition may be generated from these:

$$\mathcal{P}\rho : \mathit{wp}[R, \mathit{dur}] \ \leftrightarrow\ \rho : \mathit{wp}[R, \mathit{dur}, \mathit{true}] \tag{25}$$

$$\mathcal{S}\rho : \mathit{wp}[R, \mathit{post}] \ \leftrightarrow\ \rho : \mathit{wp}[R, \mathit{true}, \mathit{post}] \tag{26}$$

$$\rho : \mathit{wp}[R, \mathit{dur}, \mathit{post}] \ \leftrightarrow\ \mathcal{P}\rho : \mathit{wp}[R, \mathit{dur}] \ {}_\wedge\ \mathcal{S}\rho : \mathit{wp}[R, \mathit{post}] \tag{27}$$

It will be shown first that $\rho : \mathit{wp}[R, \mathit{dur}, \mathit{post}]$ is expressible in the logic.

There is a semantic denotation for the weakest precondition. It is a certain set of states. For the case of execution until termination, for example, it is the back image of the set of $W'T'$ pairs that model *post* under the relation $\mathcal{S}\rho[\![R]\!]$ given in Table I, i.e.,

$$\begin{aligned}\mathcal{S}\rho : wp[\![R, post]\!] &= \mathcal{S}\rho[\![R]\!](\!|post|\!) \\ &= \{WT | \forall W'T' | WT\mathcal{S}\rho[\![R]\!]W'T' \bullet W'T' \models post\}\end{aligned} \tag{28}$$

and for the during-conditions likewise:

$$\begin{aligned}\mathcal{P}\rho : wp[\![R, dur]\!] &= \mathcal{P}\rho[\![R]\!](\!|dur|\!) \\ &= \{WT | \forall W'T' | WT\mathcal{P}\rho[\![R]\!]W'T' \bullet W'T' \models dur\}\end{aligned} \tag{29}$$

Note that the back image of the relation is taken to be the set of WT pairs *all* of whose images satisfy *post*.

Model-theoretic and proof-theoretic weakest preconditions are distinguished notationally. The former is a set, and the latter is a formula. The definitions (28) and (29) are model-theoretic. There is a corresponding two-part weakest precondition:

$$\rho : wp[\![R, dur, post]\!] \;=\; \mathcal{P}\rho : wp[\![R, dur]\!] \wedge \mathcal{S}\rho : wp[\![R, post]\!] \tag{30}$$

We need a formula with this denotation.

LEMMA 9.1. *The model-theoretic weakest precondition $\rho : wp[\![R, dur, post]\!]$ is expressible as a temporal logic formula whenever dur and post are.*

This result is central to the proof. It goes by structural induction on R, using the semantics given in Table I. The required constructions are given in Figure 8.

Note that the weakest precondition for a loop is expressed as an infinite conjunction (see Nelson [1989]) in the figure, but, for any particular invariant *dur* and postcondition *post*, the expression is the denotation of a concrete formula: the Goedel encoding of the statement that the state machine that the code represents will terminate with the required postcondition and that while it is running the during-condition holds.

In practice, the construction is always much less exacting. All well-designed VHDL loops contain a `wait` statement in each branch, and each will wait some fixed minimum of time, say 1 ns. That means that provided the postcondition bounds the time, only a certain number of loop iterations need ever be considered. Further, the weakest precondition for loops is never needed in practice. It is needed here to show the technical existence of a certain intermediate condition for arbitrary sequences of codes that may contain loops. But well-designed VHDL code should not contain loops in sequence. Each VHDL process should contain only the single (implicit) process loop, and VHDL processes cannot be placed in sequence.

We can show that the model-theoretic weakest precondition is also the proof-theoretic weakest precondition, if there is one.

LEMMA 9.2. *If the proof-theoretic weakest precondition exists, then it coincides with the model-theoretic weakest precondition.*

Note also that the formula for the model-theoretic weakest precondition is algorithmically constructible from the syntax alone.

By inspection of each of the model-theoretic weakest preconditions in Figure 8, there is a proof in each case that shows it to be a precondition. It must then be the

$$\frac{\mathcal{S}\rho : \mathit{pre}\ \{a\}\ \mathit{mid} \quad \mathcal{P}\rho :\ \mathit{mid}\ \{b\}\ \mathit{dur} \qquad \mathcal{P}\rho : \mathit{pre}\ \{a\}\ \mathit{dur}}{\mathcal{P}\rho : \mathit{pre}\ \{a\ ;\ b\}\ \mathit{dur}} \tag{31}$$

$$\frac{}{\mathcal{P}\rho : \mathit{pre}\ \{\texttt{wait on}\ X\}\ \mathit{dur}}[\forall\rho|\mathit{pre} \bullet \exists x|\mathit{dur}\,\underline{\textbf{while}}\ \ X{=}x] \tag{32}$$

$$\frac{\mathcal{P}\rho : \mathit{pre}_{\wedge}\, x\ \{b_1\}\ \mathit{dur} \qquad \mathcal{P}\rho : \mathit{pre}_{\wedge}\, \neg x\ \{b_0\}\ \mathit{dur}}{\mathcal{P}\rho : \mathit{pre}\{\texttt{if}\ x\ \texttt{then}\ b_1\ \texttt{else}\ b_0\}\mathit{dur}} \tag{33}$$

$$\frac{\mathcal{S}\rho : I_{\wedge}\, x\ \{b\}\ I \qquad \mathcal{P}\rho : I_{\wedge}\, x\ \{b\}\ \mathit{dur}}{\mathcal{P}\rho : \mathit{pre}\{\texttt{while}\ x\ \texttt{do}\ b\}\mathit{dur}}[\forall\rho|\texttt{pre}_{\wedge}\,\texttt{x} \bullet \texttt{I}] \tag{34}$$

$$\frac{\mathcal{P}\rho : \mathit{pre}_{\wedge}\, \underline{\square}\,(\rho{=}0)\ \{\texttt{while}\ \mathit{true}\ \texttt{do}\ b\}\ \mathit{dur}}{\mathcal{P}\rho : \mathit{pre}\ \{\texttt{process}\ [\rho : \texttt{out}]\ \texttt{begin}\ b\ \texttt{end}\}\ \mathit{dur}} \tag{35}$$

$$\frac{\mathcal{P}\rho : \mathit{pre}\ \{a\}\ \mathit{dur} \qquad \mathcal{P}\rho : \mathit{pre}\ \{b\}\ \mathit{dur}}{\mathcal{P}\rho : \mathit{pre}\ \{a\,||\,b\}\ \mathit{dur}} \tag{36}$$

$$\frac{}{\mathcal{P}\rho : \mathit{pre}\ \{[\mathit{pre}'\,|\,\mathit{dur}']\}\ \mathit{dur}}[\forall\rho|\mathit{pre} \bullet \mathit{pre}'_{\wedge}\,\forall\rho|\mathit{dur}' \bullet \mathit{dur}] \tag{37}$$

$$\frac{}{\mathcal{P}\rho : \mathit{pre}\ \{\texttt{null}\}\ \mathit{dur}} \tag{38}$$

$$\frac{}{\mathcal{P}\rho : \mathit{pre}\ \{X \Leftarrow Y\ \texttt{after}\ \tau\}\ \mathit{dur}} \tag{39}$$

Fig. 10. Hoare logic for execution of VHDL statements until continuation.

weakest proof-theoretic precondition. Any weaker precondition would be a weaker model-theoretic precondition too, because the logic is sound (Proposition 8.2.4), i.e.,

LEMMA 9.3. *Both $\mathcal{S}\rho : wp[\![R, post]\!]\ \{R\}\ post$ and $\mathcal{P}\rho : wp[\![R, dur]\!]\ \{R\}\ dur$ hold for each statement or specification R.*

COROLLARY 9.4. *The proof-theoretic weakest precondition exists* (*and is the model-theoretic one and is algorithmically constructible*).

The *falsification* of a not globally valid assertion triple is defined as follows. If $\not\models \mathcal{S}\rho : \mathit{pre}\ \{R\}\ \mathit{post}$, and WT and $W'T'$ are the *witnesses*, in the sense that $WT \models \mathit{pre}$ and $WT\mathcal{S}\rho[\![R]\!]W'T'$, but $W'T' \not\models \mathit{post}$, then W, W' will be said to *falsify* $\mathcal{S}\rho : \mathit{pre}\ \{R\}\ \mathit{post}$ at T, T' respectively, and likewise for $\mathcal{P}$.

Now we present the sketch of the proof of the completeness theorem.

(1) Every proof rule has a completely general predicate pattern and a uniquely specific code/specification pattern on the bottom, and all the patterns are covered (disjointly).

(2) Every proof rule has simpler code/specifications on the top than on the bottom.

(3) Every proof rule is either Y or I shaped (or has no top). There are no alternate proofs concerning a given code/specification.

(4) The top of each rule can be generated from the bottom mechanically. In the case of the rule for sequence we can choose to use the weakest precondition of the

second in the sequence as the intermediate condition. In the case of loops, we can choose to use the invariant $\wedge F^n[post]$ (which reduces to a certain formula) from the weakest precondition semantics (this is the weakest invariant).

(5) So an entire proof tree can be generated mechanically. If all the leaves are valid axioms (in which the side-conditions are identities of temporal logic) then the assertion triple at the bottom is proved, because the logic is sound.

(6) If, on the other hand, one of the generated leaves is not an axiom, then it is a rule with nothing on top in which the side-condition *cond* fails to be a valid statement of temporal logic. Then there is a world line W and a time T such that $WT \not\models cond$.

(7) Inspection of each axiom (each of which has the form $cond \vdash pre\ \{R\}\ p$) shows that a WT pair such that $WT \not\models cond$ has W,W falsifying $pre\ \{R\}\ p$ at T and some time $T' \geq T$, respectively.

(8) Each deduction rule read downward can be seen to construct a falsification of the bottom of the rule from a falsification of (any one of) the top parts.

(9) So, if a leaf at the top of the generated proof tree is falsifiable, so is the assertion at the root, which must therefore fail to be universally valid.

(10) Hence, either an assertion is provable (and hence valid), and we have generated a proof, or it is falsifiable.

10. SUMMARY

We have presented a refinement calculus based on a semantics and programming logic for a kernel subset of the IEEE standard hardware description language VHDL. The treatment is not confined to discrete time. The refinement calculus is the first to have been set out for VHDL. The semantics is the first of its kind to be offered for VHDL. The calculus and the logic are complete with respect to the semantics.

In essence, the treatment here reduces the verification or synthesis of VHDL to a problem in temporal logic, to be settled there. In that respect, the completeness results here are relative to the underlying temporal logic.

REFERENCES

Back, R.-J. and Sere, K. 1991. Stepwise refinement of action systems. *Struct. Program. 12*, 17–30.

Borrione, D., Ed. 1995. Special issue on VHDL semantics. *Formal Methods Syst. Des. 7*, 1/2 (Aug.).

Borrione, D., Pierre, L., and Salem, A. 1992. Formal verification of VHDL descriptions in the Prevail environment. *IEEE Des. Test Comput. 9,* 2, 42–56.

Bose, B. and Johnson, S. 1993. DDD-FM9001: Derivation of a verified microprocessor. In *CHARME'93*, Lecture Notes in Computer Science, vol. 683. Springer-Verlag, Berlin, 191–202.

Breuer, P., Martínez Madrid, N., Sánchez, L., Marín López, A., and Delgado Kloos, C. 1996a. A formal method for specification and refinement of real-time systems. In *Proceedings of the 8th EuroMicro Workshop on Real Time Systems.* IEEE Computer Society Press, Los Alamitos, Calif.

Breuer, P., Martínez Madrid, N., Sánchez, L., Marín López, A., and Delgado Kloos, C. 1996b. Putting logical time into a real-time language. In *the Asia-Pacific Conference on Hardware Definition Languages.* 107–111.

Breuer, P., Sánchez, L., and Delgado Kloos, C. 1995. A simple denotational semantics, proof theory and a validation condition generator for VHDL. *Formal Methods Syst. Des.*, *7*, 1/2 (Aug.), 27–52.

Christian, F. 1994. Correct and robust programs. *IEEE Trans. Softw. Eng. 10*, 163–174.

Déharbe, D. and Borrione, D. 1995. Semantics of a verification-oriented subset of VHDL. In *Correct Hardware Design and Verification Methods*, P. Camurati and H. Eveking, Eds., Lecture Notes in Computer Science, vol. 987. Springer-Verlag, Berlin, 293–310.

Delgado Kloos, C. and Breuer, P., Eds. 1995. *Formal Semantics for VHDL*. Kluwer, Amsterdam.

Fidge, C. 1993. Real-time refinement. In *FME'93: Industrial-Strength Formal Methods*, J. Woodcock and P. Larsen, Eds., Lecture Notes in Computer Science, vol. 670. Springer-Verlag, Berlin, 314–331.

Gordon, M. 1995. The semantic challenge of Verilog. In *The 10th Annual IEEE Symposium on Logic in Computer Science*. IEEE Computer Society Press, Los Alamitos, Calif., 136–145.

Gordon, M. and Melham, T. 1993. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, Cambridge, UK.

IEEE. 1988. IEEE standard VHDL language reference manual. IEEE Std. 1076-1987. Institute of Electrical and Electronics Engineers, New York.

Johnson, S., Miner, P., and Pullela, S. 1994. Studies of the single-pulser in various reasoning systems. In *The 2nd Conference of Theorem Provers in Circuit Design: Theory, Practice and Experience*, R. Kumar and T. Kropf, Eds. Lecture Notes in Computer Science, vol. 901. Springer-Verlag, Berlin, 209–228.

Jones, C. B. 1983. Tentative steps toward a development method for interfering programs. *ACM Trans. Program. Lang. Syst. 5,* 4 (Oct.), 596–619.

King, S. and Morgan, C. 1995. Exits in the refinement calculus. *Formal Aspects Comput. 7*, 54–76.

Kopetz, H., Zainlinger, R., Fohler, G., Kantz, H., Puschner, P., and Schütz, W. 1991. The design of real-time systems: From specification to implementation and verification. *Softw. Eng. J. 3,* 6 (May), 72–82.

Kropf, T., Schneider, K., and Kumar, R. 1994. A formal framework for high level synthesis. In *The 2nd Conference of Theorem Provers in Circuit Design: Theory, Practice and Experience*, R. Kumar and T. Kropf, Eds. Lecture Notes in Computer Science, vol. 901. Springer-Verlag, Berlin, 307–323.

Kumar, R. and Kropf, T., Eds. 1994. *The 2nd Conference of Theorem Provers in Circuit Design: Theory, Practice and Experience.* Lecture Notes in Computer Science, vol. 901. Springer-Verlag, Berlin.

Mahony, B. P. and Hayes, I. J. 1991. Using continuous real functions to model timed histories. In *Proceedings of the 6th Australian Software Engineering Conference* (*ASWEC91*). Australian Computer Society, 257–270.

Mayger, E., Francis, M., Harris, R., Musgrave, G., and Fourman, M. 1991. The need for a core method. In *EuroMicro'91.*

Mermet, J., Ed. 1992. *VHDL for Simulation, Synthesis and Formal Proofs of Hardware*. Kluwer, Amsterdam.

Morgan, C. 1994. *Programming from Specifications*. Prentice-Hall, Englewood Cliffs, N.J.

Morris, J. 1987. A theoretical basis for stepwise refinement and the programming calculus. *Sci. Comput. Program. 9*, 287–306.

Nelson, G. 1989. A generalization of Dijkstra's calculus. *ACM Trans. Program. Lang. Syst. 11,* 4, 517–561.

Salem, A. and Borrione, D. 1992. Formal semantics of VHDL timing constructs. In *VHDL for Simulation, Synthesis and Formal Proofs of Hardware*, J. Mermet, Ed. Kluwer, Amsterdam.

Shaw, A. 1989. Reasoning about time in higher-level language software. *IEEE Trans. Softw. Eng. 7,* 15 (July), 875–889.

van Tassel, J. 1990. The semantics of VHDL with VAL and HOL. Tech. Rep. 196, Computer Laboratory, Univ. of Cambridge, Cambridge, UK. June.

van Tassel, J. 1993. A formalization of the VHDL simulation cycle. *IFIP Trans. A—Comput. Sci. Tech. 20*, 359–374.

WILSEY, P. 1992. Developing a formal semantic definition of VHDL. In *VHDL for Simulation, Synthesis and Formal Proofs of Hardware*, J. Mermet, Ed. Kluwer, Amsterdam.

WORDSWORTH, J. 1992. *Software Development with Z.* Addison-Wesley, Reading, Mass.

Received September 1995; revised June 1996 and January 1997; accepted February 1997