Just in Time: a new formal method for specification and refinement of real time processes

Peter T. Breuer\textsuperscript{1}, Jonathan P. Bowen\textsuperscript{2}, Natividad Martínez Madrid\textsuperscript{1}, Luis Sánchez\textsuperscript{1} and Carlos Delgado Kloos\textsuperscript{1}

\textsuperscript{1} Departamento de Ingeniería de Sistemas Telemáticos, Universidad Politécnica de Madrid, ETSI Telecomunicación, Ciudad Universitaria, E–28040 Madrid, Spain. <ptb,nnmadrid,ckdk@dit.upm.es> tel. +34-1-549-5700 x442

\textsuperscript{2} Oxford University Computing Laboratory, Programming Research Group, Wolfson Building, Parks Road, Oxford OX1 3QD, UK. <bowen@comlab.ox.ac.uk> tel. +44-865-283512

Abstract

A new specification and refinement calculus for real time processes is set out here. Under \textit{JiT} ("jet"), a specification is a three-phase "before, during and after" set of temporal assertions, connected by cross-constraints. Specifications laid out on the page have the look of an Alice-in-Wonderland wardrobe, with a door to enter, a door to peek out of, and a door to exit by, plus a bottom drawer full of undergarments. The specification style is reminiscent of Z and VDM and is intended to appeal to the same technologists. Refinements are pseudo-code containing scheduling and wait statements.

Keywords Formal verification, formal specification, refinement calculus, real-time systems, program logic and semantics, software engineering.

What I say three times is true
– Lewis Carroll (1832-1898)

1 Introduction

This paper aims to describe a specification and refinement method for real-time processes with a solid theoretical basis. In contrast to other approaches, this paper sets out a technique, called \textit{JiT}, whose pedigree is derived from the synthesis of system-level \textit{hardware} descriptions from higher-level specifications. The genesis of the technique is as follows: there have been efforts in recent years to put the semantics of the IEEE standard hardware specification language VHDL on a strictly formal footing. \cite{4} is the latest widely available text on the area, and it sets out various approaches to the semantics of the language, via explicitly timed Petri nets, statechart-based analysis, evolving algebras and so on. Declarative semantics of the language given in that reference have been shown to have a complete first order logical axiomatization \cite{2}. That axiomatization in turn has recently been shown to give rise to a complete formal refinement calculus with respect to general temporal specifications. Leaving out the portion particular to the atomic code statements of the language gives a specification and a refinement calculus for all real-time processes.

The calculi are based on three-phase specifications: the conditions under which a real-time process may be \textit{entered}, the conditions that hold \textit{during execution}, and the conditions that hold when or if the process \textit{terminates}. A specification in this formalism has been likened to a “magic wardrobe” with three doors, where entry is through one door, throughout the journey one may
peek out of the middle door, but not exit, and at the end of the journey the real-time process may leave by the final door.

This style of specification is related to the pre-/post-condition semantics famously introduced by Hoare, but adds a third condition between the other two. It has obvious meaning for hardware components, in that most hardware components endure for a significant time, so it is important not only to specify what happens at termination, but to specify the behaviour that persists until they cease to exist! An adder adds until the power switch is flicked off.

On the other hand, a continuous process, whether software or hardware, may still be regarded as being composed of infinitely many finitely enduring subprocesses. The adder, for example, may operate in discrete phases: data acquisition, calculation and output of results, repeated ad infinitum. Whilst it is possible to decompose the adder into parallel persistent and communicating processes, it makes more sense to design it as a simple iterator of these three finite phases.

In this context it becomes important to know about the termination and initiation conditions of each phase too, as well as the conditions that hold whilst the phase exists. Moreover, even if the adder is decomposed into, say, a data acquisition latch, arithmetic logic circuitry and a data output buffer (three parallel persistent units), the operation of each of these can still be described conveniently as a repeated cycle. The input buffer receives a signal, buffers it and waits for the next signal. The logic circuitry waits for a change notification from the input buffers and signals the output buffers with the appropriate results. The output buffers cycle between waiting for a signal from the logic circuitry and fixing it in their store. So describing termination is important even for continuously running processes, because a continuously running process iterates through many terminating phases.

In what way is the formal method set out here different from other formal methods? Chiefly in its offering of a refinement calculus for temporal specifications. The specification format has the look of $Z$ [15], and there is indeed a schema calculus, like that of $Z$, which allows specifiers to refine specifications to a point where they may be implemented. The schema calculus is a subset of the full refinement calculus (which is based on a pair of weakest precondition semantics for logics of program termination and suspension), but it is all that is necessary in order to make and manipulate and reason about the design of real-time processes. We believe that there is no closely related formal method in existence, and that the method, being rooted in the real needs of hardware description languages, can serve the needs of real-time system designers.

The layout of this article is as follows. After the present section, and the following one on the background to the area, Section 3 introduces the specification and refinement technique through a simple worked example. Section 4 sets out some model-theoretic background for those who care to know, and Section 5 discusses the applicability of the technique and makes comparisons with other formal techniques for real-time specification.

2 Background

Solid theoretical support for the practice of real-time programming has a much shorter history than that of its cousins supporting the development of programs for off-line libraries and applications. As far as formal methods for the latter area is concerned, it must be acknowledged that the industry has viable technologies [7] and that human, managerial and commercial factors are the governing influences on the future spread and acceptability of the science.

With respect to formal methods for real-time programming [10], an obvious but superficial “upgrade path” from formal methods for un-timed programs is to treat time as a read-only shared variable, and then proceed as for un-timed programs with the addition of this special variable [14]. This idea has been part of the folklore in the formal methods world for a good while, and it certainly works at the specification level, because saying that an operation should be completed “within 5ms of the request being received”, for example, is the natural way of expressing the user-level constraints that are practically of interest.

But progression from the highest level of specification to lower levels and code can be difficult via this technique. Dealing directly with a variable at the top level – albeit representing as tangible
a concept as time – requires one eventually to implicitly or explicitly constrain the action of every elementary operation or sub-specification with respect to it. That may be no bad thing, because, after all, time is advanced by every operation, but, nevertheless, better treatments are being sought. In any case, the awkwardness of expressing a concept such as “no change for 1ms” in terms of elementary variables makes some kind of higher-level abstraction desirable.

As distinct from the explicitly-quantified time kind of specification, specifications making use of some form of temporal propositional logic (that is, a modal logic of time) have been notching up significant successes using model checking techniques [3] over the last ten years, primarily due to the efficiency of the algorithms for checking implementations against specifications couched in this propositional style. For finished finite state machines such as hardware microprocessors the science is solid (although the complexity is daunting), but a strategy for getting to a machine description from an abstract specification may not be as clear as the strategy for verifying an existing description. Nevertheless, hardware compilation and verification may be approaching a mature stage through this science [1].

Statecharts (timed state transition diagrams) [9] typify another promising school. The technique holds out the reality of a development path from specification to implementation via the progressive refinement of timing diagrams. The very visual representation style is popular, but may make it too easy to program operationally rather than specify what is wanted in declarative style; a paradoxical situation, since the elementary expressions are temporal formulae, which could not be more declarative. Time constraints on transitions are usually specified as intervals in which the transition must occur, which ties up with interval temporal logics and other wrappers that can be related back to the explicit-time school to give more structure to upgraded forms of standard formal methods.

There is a lot of academic activity in adding time (interval logic, temporal logic, Duration Calculus, etc.) to basic specification languages such as Z and wide-spectrum specification and refinement languages such as VDM (e.g., see [5, 6, 13]). The ongoing effort is at least as great as that involved in adding object-oriented flavours to the basic stock [12].

Recently, there has been consideration of hybrid systems [8] in which continuous control variables as well as a continuous time variable is considered. This is especially important in embedded systems where a digital computer attempts to control its analog environment. The requirements for such a system are necessarily in the domain of continuous mathematics (differential equations, etc.) whereas the computer-based implementation is in the domain of discrete mathematics (predicate logic, etc.). The process of producing a specification, design, program, compilation and consideration of the hardware itself must span and link the various levels of abstraction in some credible manner [11].

3 Specification

Consider the simple task of designing 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 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:

\[
Q = T \mod 2
\]

At startup time \( T = 0 \) no particular initial conditions will be imposed, only the time will be specified. Contrariwise, no particular 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 then. It must take the value \( Q = T \mod 2 \).

It may be imagined that the process is being momentarily suspended from time to time in order to look at its state and its output. The startup time requirement \( T = 0 \) and the requirement \( Q = T \mod 2 \) at the suspension times will be referred to as pre- and during-conditions respectively. In addition, to express the idea that the oscillator should never stop running, we add the post-
The three conditions above form a specification triple. The following will be the top-level formal specification of the oscillator for which we eventually want to derive a concrete implementation:

\[
\begin{array}{|c|c|c|}
\hline
\text{Oscillator} & \text{T' = 0} & Q = \text{mod} 2 \\
\hline
\text{Q : \text{false}} & \text{false} & \text{false} \\
\hline
\end{array}
\]

The backwards prime on the \( T \) in the first compartment of this “wardrobe” specification indicates that it is a value of a variable on entry to the process. The time \( T \) at which it would finish, if it only could, would be denoted by a forward prime.

That the process \( a \) should implement this specification will be written:

\[ \left[ Q : \text{false} \cdot \text{T = 0} \mid Q = \text{mod} 2 \mid \text{false} \right] \subseteq a \]

in linear form where \( \subseteq \) is the refinement relation. The relation is transitive and all constructions of code and specifications 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 \text{while true do . . .}:

\[ a \subseteq b \text{ and } b \subseteq c \text{ implies } a \subseteq c \quad a \subseteq b \text{ implies } f[a] \subseteq f[b] \]

Specifications, processes and code statements all lie in the domain of this relation. A specification may refine a specification; a code may refine a specification; a code may (only trivially) refine a code, and so on.

The development of the real-time process \( a \) from the specification \( \left[ Q : \text{false} \cdot \text{T = 0} \mid Q = \text{mod} 2 \mid \text{false} \right] \) will consist of a series of steps:

\[ \left[ Q : \text{false} \cdot \text{T = 0} \mid Q = \text{mod} 2 \mid \text{false} \right] = s_0 \subseteq s_1 \subseteq . . . \subseteq s_{n-1} \subseteq s_n = a \]

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

### 3.1 Initialization

Suppose that the process begins at time zero and that we are going to reset the output to zero at that time, as an initialization, before anything else happens. The reset takes no time, and it is not possible to interrupt it, but it does terminate. When it terminates, the process will have set the output line to zero and will also have scheduled the outputs to be zero forever, which will be written \( \Box \text{Q' = 0} \):

\[
\begin{array}{|c|c|c|}
\hline
\text{Reset} & \text{T' = 0 and } \Box \text{Q' = 0} \\
\hline
\text{T = 0} & \text{false} & \text{false} \\
\hline
\end{array}
\]

Under those circumstances, the oscillator can be designed as a sequence

\textbf{Oscillator is Reset \text{\&} Oscillator process proper}
in which the secondary oscillator process design can be eased by knowing the state of the output line, and that no changes are scheduled on that line:

\[
\begin{array}{c|c|c|c}
\text{Oscillator process proper} & \text{'}T = 0 \text{ and } \square \text{'}Q = 0 & Q = T \text{ mod } 2 & \text{false}
\end{array}
\]

3.2 Iteration

Suppose that the oscillator process proper is to be implemented as a single process. Then it will have the form of a while loop. Because the oscillator can never terminate, so the loop will loop forever. It will have the form \texttt{while true do ... od}. To specify the interior of the loop, a \textit{loop invariant}, \(I\), will be needed.

The invariant must hold at the beginning of each cycle, at the end of each cycle, and during each cycle. Moreover, it must be set up on the first entry to the loop, and must be sufficiently strong to maintain the oscillator invariant that we are interested in:

\[\text{'}T = 0 \text{ and } \square \text{'}Q = 0 \implies I \implies Q = T \text{ mod } 2\]

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

\[
\begin{array}{c|c|c|c}
\text{Oscillator loop body} & I \text{ and } \text{'}T = t_n & I' \text{ and } T' = t_{n+1}
\end{array}
\]

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

\(t_n = n\)

and let the invariant be such as to force the output \(Q\) to be the proper value \textit{for the remainder of the current unit time interval}. Here, \(\texttt{p for } \tau\) is the temporal logic construction that asserts that a condition \(p\) is currently holding and is planned to endure for a time \(\tau\):

\(I\) is \(Q = T \text{ mod } 2\) \texttt{for } \((1 - T \text{ mod } 1)\)

The oscillator process proper should then have the form:

\texttt{while true do Oscillator loop body done}

and all will be well. All that has to be checked is that this choice of invariant does indeed force the oscillator invariant \(Q = T \text{ mod } 2\) whenever we choose to peek (it does, because \(\texttt{p for } \tau\) implies \(p\)), and that it is forced by the oscillator process proper initial condition \(T = 0 \text{ and } \square Q = 0\) (it is, because \(\square Q = 0\) implies \(Q = 0\) in particular).

3.3 Sequence

Prefixing the oscillator process proper with a reset operation was an example of decomposing a process into a sequential composition of specifications (processes). We want to do it again for the oscillator loop body. We can imagine that the loop body first schedules a change in the output at the appropriate future time, then waits for that change to occur:

\(\text{Oscillator loop body is Schedule; Wait}\)

This is the refinement rule:
Law 1 (Sequence)

\[ [\text{pre} \mid \text{dur} \mid \text{post}] \subseteq [\text{pre} \mid \text{dur} \mid \text{mid}] \bowtie [\text{mid} \mid \text{dur} \mid \text{post}] \]

and in this case, since the scheduling part takes place at the beginning of each cycle, we can design it to be un-interruptible and to schedule a change in one unit of time’s time. Let \( q \) be a logical constant that captures the initial value of the output \( Q \). Suppose that the scheduler begins to fire at a time when \( Q \) is scheduled to keep this value for at least one unit of time (\( Q = q \text{ for } 1 \)). Then it can schedule an inversion after that one unit of time has passed (using the temporal logic construction \( p \text{ then } q \) for a condition \( q \) that holds immediately after condition \( p \) has passed, \( Q = q \text{ for } 1 \text{ then } \square Q = \neg q \), and no other changes in the schedule:

<table>
<thead>
<tr>
<th>Schedule</th>
<th>( Q = q \text{ for } 1 \text{ false} )</th>
<th>( Q' = q \text{ for } 1 \text{ then } \square Q' = \neg q )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( T = T' = n )</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The “peek” condition being \text{false} effectively forces the termination time to be the same as the start time, but there is no harm in emphasizing it with \( \langle T = T' = n \) (for the \( n \)th loop iteration). Here it is implicit that the strong condition \text{false} implies any other condition that might be required (there is a law of \textit{weakening} of specifications that should be made explicit in a longer essay).

The wait part of the process can now start off assuming that (at time \( T = n \)) a change has been scheduled in one unit of time, and wait for the change to occur:

<table>
<thead>
<tr>
<th>Wait</th>
<th>( Q = q \text{ for } 1 \text{ then } \square Q = \neg q )</th>
<th>( Q = q \text{ for } (1 - T \mod 1) \text{ then } \square Q' = \neg q )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( T = n \text{ and } T' = n + 1 )</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

In most real-time processing languages, both these specifications are implementable immediately. The schedule operation may be written:

\[ \text{schedule } Q := \neg Q \text{ after } 1 \]

and the wait operation may be written

\[ \text{wait on } Q \]

which gives the oscillator the shape

\[ \text{init } Q = 0 \text{ while true do schedule } Q := \neg Q \text{ after } 1 \text{ wait on } Q \text{ od} \]

and this implementation has been proved to satisfy its specification in the course of the development and refinement of the specification.

4 Semantic Domains

The denotation of processes and specifications will lie in the same domain. They are relations between world lines with given current time pointers. A world line \( W : \text{WorldLine} \) is a record of the past and also the scheduled future development of the system. It is a map from times to system states, which are bindings of (partial functions to) values with signal and variable names:

\[
\text{WorldLine} = \text{Time} \rightarrow \text{State} \\
\text{State} = \text{Id} \rightarrow \text{Value}
\]
A distinguished current time $T : \text{Time}$ will be associated with each world line in a process denotation. It is not necessary for the purpose of this exposition to further specify the time domain, but integer, rational, real and non-standard integer, rational or real lines are all possible and suitable. In particular, the semantics to be presented below makes no assumption over whether the time domain is discrete or continuous, linear or branching. In any $t$th world $W_t$ of a world line $W$ the reserved variable $T$ takes the value $t$, which is written as follows:

$$W_t \models T = t$$

In a time zero world, time $T$ has the value 0, in a time 1 world it has the value 1, and so on. These worlds may either be historical, in that the current time associated with the world line may be greater than their time $T$; or they may be scheduled as future states of the system, which is the case when the current time associated with the world line is less than their time $T$. If the current time associated with the world line is the same as the time $T$ in a world $W_T$, then that world is the present time world.

The generic specification $[\text{pre} | \text{dur} | \text{post}]$ denotes a pair of relations (here $A \leftrightarrow B$ is the type of relations with domain $A$ and codomain $B$:

$$[\text{pre} | \text{dur} | \text{post}] : \text{Semantics}$$

$$\text{Semantics} = (\text{WorldLine, Time}$ $\leftrightarrow \text{WorldLine, Time}) \times (\text{WorldLine, Time}$ $\leftrightarrow \text{WorldLine, Time})$$

that connects pairs ‘$W \cdot T$’ satisfying a pre-condition $\text{pre}$, to pairs $W_T$ satisfying the during-condition $\text{dur}$, and pairs $W' \cdot T$ satisfying the post-condition $\text{post}$. The concept is that a process is being specified that starts at a time ‘$T$’ in a state described by ‘$W \cdot T$’ and satisfying $\text{pre}$. That is:

‘$W \cdot T \models \text{pre}$

At that moment, the process has a history corresponding to the tail of the world line ‘$W$’ at times more ancient than ‘$T$’, and a schedule of (or plan for) future events that corresponds to the front segment of the world time ‘$W$’, at times later than ‘$T$’.

When the process is allowed to begin execution and is later momentarily suspended at a time $T$, then it will be in the state $W_T$ of the world line $W$, which should satisfy $\text{dur}$:

$W_T \models \text{dur}$

At that moment it will have a history corresponding to the tail of the world line $W$ (the worlds at times older than $T$), and a schedule of future events that corresponds to the front segment of the world time $W$ (the worlds at times later than $T$).

When the process terminates at a time $T'$, then it will be in the state $W' \cdot T'$ of the world line $W'$, which should satisfy $\text{post}$:

$W' \cdot T' \models \text{post}$

A causal process will behave in such a way that the history at suspension or termination is at least a continuation or a preservation of the history at earlier times, but it is quite possible to make necessarily a-causal specifications:

$$[Q : (T = 0 \land Q = 0) \land (T \geq 0) \land \text{“Q has never been 0”} \land \text{false}]$$

is one such. In this specification, the future influences the past. There is no possible implementation of this specification as a causal process. Most of the specifications that are drawn up permit acausalities, because they do not specify explicitly that, for example, time must move on, history be preserved, and other necessities. If the specification is refinable to an implementation, however, then it admits of a causal solution, and this is the case of interest.
5 Discussion

To be discussed . . .

6 Conclusion

A specification and refinement calculus for real-time processes has been set out here. The calculi are solidly based on a “three-phase semantics and logic that associates to each process or fragment of a process a pre-condition, an invariant and a post-condition. These hold true respectively on entry to the process, while it is running, and on exit from the process. Processes that never terminate can be expressed through a false post-condition.

The method has been introduced via illustrative examples and its applicability and scalability have been discussed. We believe that this calculus will provide a secure and scalable foundation for the application of formal methods to real-time processes.

References


