

Modelling real-time multi-tasking systems with timed traces*

C. J. Fidge
Software Verification Research Centre
Department of Computer Science
The University of Queensland
Queensland 4072, Australia

Version 2.0, January 1998

Abstract

We formalise the behaviour of non-preemptive, real-time, multi-tasking systems by expressing the computational components assumed by real-time scheduling theory in a trace-based notation. The model is suitable as a target implementation domain for dataflow refinements, amenable to formal schedulability analysis, and implementable in a concurrent real-time programming language.

1 Introduction

We present a formal model of real-time, static-priority, non-preemptive process scheduling. The model

- is trace-based, making it a suitable target domain for formal ‘dataflow’ refinements [9],
- is expressed using the computational components assumed by real-time scheduling theory, making it amenable to analysis via an already-proven real-time schedulability test [1],

*An earlier version of this report was published in *Proc. Third Australasian Conference on Parallel and Real-Time Systems*, Brisbane, pp. 94–100, September 1996.

- has a direct implementation in the Ada 95 programming language [3], and
- is sufficiently simple to have a good chance of acceptance in safety-critical applications [2].

2 Background

2.1 Scheduling theory terminology

A real-time, non-preemptive, multi-tasking system consists of n application *processes*. Each process has a unique static *priority*. After system initialisation is completed, at time ‘0’, each process *arrives* infinitely often. Each process i is *periodic*, with each arrival separated by T_i time units. At each arrival an *invocation* of process i must be performed before a fixed *deadline* D_i elapses, measured from the arrival time. Each invocation of process i may require up to C_i units of processor time, its worst-case *computation time* [1].

Whenever no application process is running, and one or more arrives, a run-time *scheduler* decides which process to *release* next, based on their priorities. The scheduler may require up to S_i time units to recognise that process i is ready to be released (measured from the time process i arrives) and a further S_{res} time units to *resume* execution of the chosen process. Once released, an invocation of process i executes without preemption until it voluntarily *suspends* itself. The act of suspending a process may take up to S_{sus} time units [3]. Control is then returned to the scheduler.

Release of process i may initially be *blocked* by a lower priority process which is already running when i arrives. Similarly, i may suffer *interference* from higher-priority processes. Processes communicate using *shared variables*. Since there is no preemption, mutually-exclusive access to such variables is automatically guaranteed [3].

2.2 Timed specifications

Our specifications are expressed as predicate transformers using *timed refinement calculus* [9] notation. A *specification statement* consists of three parts.

$$+\vec{c}: [A, E]$$

The *frame* vector \vec{c} defines those variables *constructed* by this specification. The *assumption* predicate A defines the anticipated environment in which the specification will work correctly. A may not refer to \vec{c} . The *effect* predicate E defines the desired behaviour that the specification achieves. Typically E defines \vec{c} .

Each constructed variable c is represented as a *timed trace*, i.e., a function from the absolute time domain \mathbb{A} to some type T , thus specifying the behaviour of c over all time.

$$c : \mathbb{A} \rightarrow T$$

Specification statements can be *composed* in parallel, where their composition is formally defined via the convergent iterative solution of their effect formulæ [10].

$$+ \vec{c}_1 : [A_1, E_1] \parallel + \vec{c}_2 : [A_2, E_2]$$

2.3 Notation for time intervals

We use the following simplified subset of the Z-based mathematical notation defined by Mahony, Hayes, et al [11, 4] for expressing trace-based specifications.

Let the absolute time domain \mathbb{A} , and durations of time \mathbb{D} , be continuous, as modelled by the reals.

$$\begin{aligned} \mathbb{A} &== \mathbb{R} \\ \mathbb{D} &== \mathbb{R}_+ \end{aligned}$$

We represent an open interval from time a to time z as follows.

$$a \circ\circ z == \{t : \mathbb{A} \mid a < t < z\}$$

To avoid frequent explicit references to the current time, let a *timed history* predicate be a predicate on a number of timed variables \vec{v} without explicit time dereferencing. We can then determine those instants t when such a predicate TH is true by replacing each ' v ' in TH with ' $v(t)$ ' [11].

$$times(TH) = \{t : \mathbb{A} \mid t \in \text{dom } \vec{v} \wedge TH \left[\frac{\vec{v}(t)}{\vec{v}} \right]\}$$

An *interval discriminator* defines the set of non-empty intervals during which a timed history predicate is true. For instance, all open intervals

during which some predicate TH is true, with the right-hand end maximal, is defined as follows [4].

$$\begin{aligned} (\neg TH \dashv) & == \\ & \{a, z : \mathbb{A} \mid a < z \wedge a \circ \circ z \subseteq \text{times}(TH) \wedge z \notin \text{times}(TH) \bullet a \circ \circ z\} \end{aligned}$$

Similarly for other bracket combinations.

Logic-like operators can be defined over interval discriminators ID , using set operators [11].

$$\begin{aligned} ID_1 \textbf{ and } ID_2 & == ID_1 \cap ID_2 \\ ID_1 \textbf{ or } ID_2 & == ID_1 \cup ID_2 \\ ID_1 \textbf{ whenever } ID_2 & == ID_1 \supseteq ID_2 \end{aligned}$$

A number of interval discriminator modifiers are available [4], of which we use only a few here. Each begins with a set of intervals, from a to z , defined by some discriminator ID . In some cases a subscript d specifies the duration of intervals from ID , and moves one of the endpoints.

$$\begin{aligned} \text{initially}_d ID & == \\ & \{a, z, z' : \mathbb{A} \mid a \circ \circ z \in ID \wedge z - a = d \wedge z \leq z' \bullet a \circ \circ z'\} \\ \text{finally}_d ID & == \\ & \{a, z, a' : \mathbb{A} \mid a \circ \circ z \in ID \wedge z - a = d \wedge a' \leq a \bullet a' \circ \circ z\} \end{aligned}$$

In other cases d specifies how far to move the endpoints.

$$\begin{aligned} \text{after}_d ID & == \\ & \{a, z, a', z' : \mathbb{A} \mid a \circ \circ z \in ID \wedge a - a' = d \wedge a' < z' \bullet a' \circ \circ z'\} \\ \text{preceding}_d ID & == \\ & \{a, z, a', z' : \mathbb{A} \mid a \circ \circ z \in ID \wedge a' - z = d \wedge a' < z' \bullet a' \circ \circ z'\} \end{aligned}$$

For the purposes of this paper we also define two operators that allow us to refer to interval durations. Firstly, all open intervals of length d are defined as follows.

$$\text{dur}_d == \{a, z : \mathbb{A} \mid z - a = d \bullet a \circ \circ z\}$$

Secondly, we want to find the total time for which some property holds in an interval. Let *covers* define all open intervals that are included in interval

discriminator ID and lie in some set of times A , such that there is no longer interval B with the same properties.

$$\text{covers}(ID, A) = \{a, z : \mathbb{A} \mid a \circ\circ z \in ID \wedge a \circ\circ z \subseteq A \\ \wedge (\nexists B : ID \bullet B \subseteq A \wedge a \circ\circ z \subset B)\}$$

Note that covers returns pairs consisting of the infimum a and supremum z of such intervals, rather than the intervals themselves. Then we define all intervals for which the total duration of time ‘covered’ by intervals from ID is d as follows.

$$\text{totaldur}_d ID == \\ \{a, z : \mathbb{A} \mid \sum_{X \in \text{covers}(ID, a \circ\circ z)} (\text{second } X - \text{first } X) = d \bullet a \circ\circ z\}$$

In each definition above the “ $= d$ ” test can be overridden by including an alternative comparator in the subscript. Omitting the subscript entirely admits any value of d [11].

The variables in our model are all imaginary and of discrete types. Their behaviour over time can be defined as continuous functions that change value only by becoming undefined at some point [11, 4].

$\begin{array}{l} \text{var} : \mathbb{P}(\mathbb{A} \leftrightarrow T) \\ \forall f : \mathbb{A} \leftrightarrow T \bullet \\ f \in \text{var}[T] \Leftrightarrow \\ \quad \forall a, z : \mathbb{A} \mid a \circ\circ z \subseteq \text{dom } f \bullet \#f(\mid a \circ\circ z \mid) = 1 \\ \quad \wedge \forall x : \mathbb{A} \setminus \text{dom } f \bullet \\ \quad \quad \exists d : \mathbb{D} \mid d > 0 \bullet (x - d \circ\circ x + d) \setminus \text{dom } f = \{x\} \end{array}$

Thus, in a contiguous interval from a to z where var function f is defined, f has only a single value— f must exhibit a ‘flat’ graph. Also, any point x where f is undefined is separated by some non-zero duration d from any other such point—all state changes must be instantaneous.

3 A real-time multi-tasking model

Using the above notations we construct a model of a multi-tasking system as the composition of a number of process and shared variable specifications,

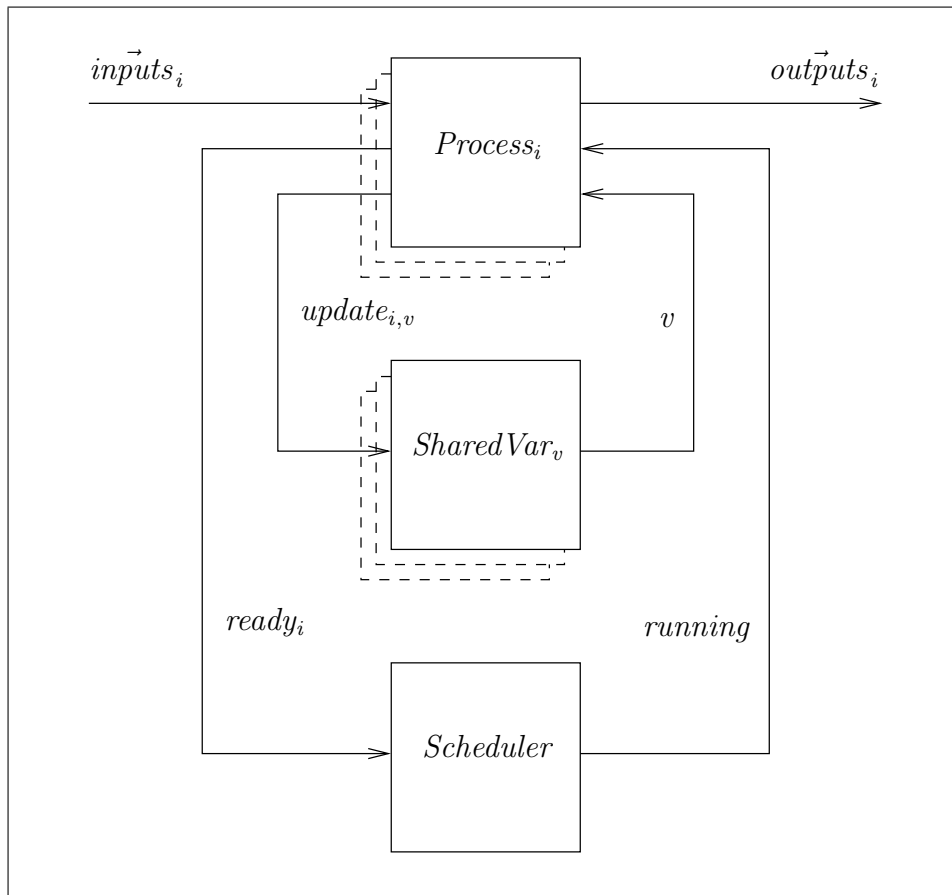


Figure 1: Overview of the model. Boxes denote parallel specification components. Arcs denote timed variables.

and a single scheduler specification (Figure 1).

$$\begin{aligned} \text{System} &\stackrel{\text{def}}{=} (\parallel_{i \in I} \text{Process}_i) \\ &\parallel (\parallel_{v \in V} \text{SharedVar}_v) \\ &\parallel \text{Scheduler} \end{aligned}$$

I and V are indexing sets identifying the processes and shared variables, respectively. Values in I have a total ordering from lowest priority process 1 to highest priority process n . Let the scheduler itself be denoted process 0, and I_0 be $I \cup \{0\}$. For some variable v from V , let I_v denote the subset of application processes that access variable v . For some process i from I , let V_i be the set of shared variables accessed by i .

3.1 Constructed variables

Interaction between these components is represented by several constructed variables. Each process i constructs a number of application-specific output variables, and reacts to a number of application-specific inputs from the environment.

$$\begin{aligned} \text{Inputs}_i &\hat{=} [\vec{\text{inputs}}_i : \mathbb{A} \leftrightarrow \dots] \\ \text{Outputs}_i &\hat{=} [\vec{\text{outputs}}_i : \mathbb{A} \leftrightarrow \dots] \end{aligned}$$

The value of a shared variable v , of type T_v , may be observed by a number of processes. Some process i may ‘update’ v by exhibiting a new value for it. Let type T_v^\perp be $T_v \cup \{\perp\}$, where special value \perp is used to indicate that no update is in progress.

$$\begin{aligned} \text{Value}_v &\hat{=} [v : \mathbf{var}[T_v]] \\ \text{Update}_i &\hat{=} \bigwedge_{v \in V_i} [\text{update}_{i,v} : \mathbf{var}[T_v^\perp]] \end{aligned}$$

The scheduler constructs a schedule of running processes from I_0 . Each process i explicitly indicates those intervals in which it is ready to run.

$$\begin{aligned} \text{Readiness}_i &\hat{=} [\text{ready}_i : \mathbf{var}[\mathbb{B}]] \\ \text{Schedule} &\hat{=} [\text{running} : \mathbf{var}[I_0]] \end{aligned}$$

3.2 Process specifications

Each period for process i is a multiple of T_i from start time ‘0’. Let period_i be the (maximal) period intervals. There are then two requirements. The

process must be ready at the start of every period. Also the process may *become* ready only at the start of a period, except when the preceding ready interval overran, in which case the process becomes ready again immediately.

$\text{ReadyPeriodically}_i$ <hr style="border: 0.5px solid black;"/> Readiness_i <hr style="border: 0.5px solid black;"/> $\text{let } period_i == \{n : \mathbb{N} \bullet (n * T_i) \circ ((n + 1) * T_i)\} \bullet$ $\quad \text{initially } (\neg ready_i) \text{ whenever } period_i$ $\quad \text{after}_0 period_i \text{ or preceding}_0 (\neg ready_i) \text{ and dur}_{>T_i}$ $\quad \text{whenever } (\neg ready_i)$

Although allowing for the possibility of overruns makes this requirement complex, it makes the overall specification below robust enough to allow for process i being denied access to the processor indefinitely (see Section 5).

$\text{ReadyPeriodically}_i$ places no limit on the duration of each ready interval. This is done below by considering the process's obligation not to consume too much processor time. At each invocation the process must not be ready and running for more than C_i time units.

$\text{WorstCaseExecTime}_i$ <hr style="border: 0.5px solid black;"/> $\text{Readiness}_i; \text{ Schedule}$ <hr style="border: 0.5px solid black;"/> $\text{totaldur}_{\leq C_i} (\neg running = i) \text{ whenever } (\neg ready_i)$
--

(The `totaldur` modifier is robust enough to allow for the possibility that process i is preempted, although we never expect this to happen.)

Lastly, each process specification i may assume certain application-specific properties about the inputs and shared variables it uses. Let these be described in predicate Environment_i . Process i must also achieve some application-specific effect on the output variables, and perform application-specific updates to shared variables. Let these be specified in predicate ProcReq_i . In particular, we expect ProcReq_i conforms with the restrictions that process i may change output variables $\vec{outputs}_i$, and issue shared variable updates using $update_{i,v}$ variables, only when it is both ready and running. (There is nothing to stop a programmer from specifying a process that does not obey these characteristics, but such a specification will not be

refinable to executable sequential code [12].)

$$Process_i \stackrel{\text{def}}{=} +\begin{matrix} \vec{outputs}_i, \\ \vec{ready}_i, \\ \vec{update}_{i,v} \end{matrix} : \left[\begin{array}{ll} \textit{Schedule} & \textit{ReadyPeriodically}_i \\ \textit{Environment}_i & \textit{WorstCaseExecTime}_i \\ & \textit{ProcReq}_i \end{array} \right]$$

No special assumption is needed about *running* due to the robustness built into requirement *ReadyPeriodically*_{*i*}.

3.3 Shared variable specifications

The value of a shared variable *v* at any time is determined by the most recent update performed, if any. In any interval when no process *j* is performing an update, and there was an immediately preceding interval in which some process *i* issued an update of value *u*, and no other process *k* simultaneously issued an update, then the value of *v* must be *u*.

$$\frac{\textit{MergeUpdates}_v}{\textit{Value}_v; \bigwedge_{i \in I_v} \textit{Update}_i} \\ \bigwedge_{i \in I_v; u \in T_v} (\neg v = u \rightarrow \mathbf{whenever} \\ \quad (\mathbf{and}_{j \in I_v} (\neg \textit{update}_{j,v} = \perp \rightarrow)) \\ \quad \mathbf{and} \\ \quad \textit{preceding}_0(\bigvee \textit{update}_{i,v} = u \rightarrow) \\ \quad \mathbf{and} \\ \quad (\mathbf{and}_{k \in I_v \setminus \{i\}} (\neg \textit{update}_{k,v} = \perp \rightarrow)))$$

The value of *v* is unspecified if an update is in progress, before the first update is performed, or if two updates overlap in time.

A shared variable specification thus guarantees to merge all updates, as long as they occur in mutual exclusion.

$$\textit{SharedVar}_v \stackrel{\text{def}}{=} +v : \left[\bigwedge_{i \in I_v} \textit{Update}_i, \textit{MergeUpdates}_v \right]$$

This specification requires each *SharedVar*_{*v*} component to react instantaneously to updates. This is not a problem, however, because the implementation of this component (Section 4.2) involves no executable code. *SharedVar*_{*v*} is merely a modelling artifact that allows several specification statements to jointly construct the same variable.

3.4 Scheduler specification

The scheduler may start a process i running only if there was some time in the last $S_i + S_{res}$ time units at which i was the highest priority ready process and no other application process was running.

$\frac{\textit{ResumeReadyProcs}}{\textit{Schedule}; \bigwedge_{i \in I} \textit{Readiness}_i}$
$\mathbf{let} \textit{highest} == \lambda t : \mathbb{A} \bullet \max(\{0\} \cup \{i : I \mid \textit{ready}_i(t)\}) \bullet$ $\bigwedge_{i \in I} \mathbf{preceding}_{<S_i+S_{res}} (\textit{highest} = i \wedge \textit{running} = 0) \rightarrow$ $\mathbf{whenever} (\neg \textit{running} = i) \rightarrow$

If a number of processes arrive in rapid succession we guarantee only that the chosen process will have been the highest priority ready process at *some* point in the recent past. This models *release jitter* in the scheduler implementation [1].

When there is a process that has been ready for the worst-case time required to schedule it, then *some* process must be running [8].

$\frac{\textit{NoUnnecessaryIdling}}{\textit{Schedule}; \bigwedge_{i \in I} \textit{Readiness}_i}$
$\bigwedge_{i \in I} \mathbf{finally} (\neg \textit{running} \in I) \rightarrow$ $\mathbf{whenever} \mathbf{initially} (\neg \textit{running} = 0) \rightarrow$ $\mathbf{and} (\neg \textit{ready}_i) \mathbf{and} \mathbf{dur}_{\geq S_i+S_{res}}$

When the scheduler starts a ready process i running, it must let i continue to run as long as i remains ready.

$\frac{\textit{NoPreemption}}{\textit{Schedule}; \bigwedge_{i \in I} \textit{Readiness}_i}$
$\bigwedge_{i \in I} (\neg \textit{running} = i) \rightarrow$ $\mathbf{whenever} (\neg \textit{ready}_i) \mathbf{and} \mathbf{initially} (\neg \textit{running} = i) \rightarrow$

When a currently running process i suspends itself, i.e., becomes not ready, then i may not continue running for more than S_{susp} time units [8].

$\frac{\textit{SuspendNonReadyProcs}}{\textit{Schedule}; \bigwedge_{i \in I} \textit{Readiness}_i}$
$\bigwedge_{i \in I} \mathbf{dur}_{\leq S_{susp}} \mathbf{whenever} (\neg \textit{running} = i \wedge \neg \textit{ready}_i) \rightarrow$

(The model makes no guarantee to perform a context switch if process i overruns, i.e., if i leaves less than S_{susp} time units before its next arrival. The same process may continue running even if it is no longer the highest priority ready process.)

The scheduler does not need to make any special assumptions about the readiness indicators. It merely reacts to whatever behaviour they exhibit.

$$Scheduler \stackrel{\text{def}}{=} +running: \left[\begin{array}{l} \bigwedge_{i \in I} Readiness_i, \\ ResumeReadyProcs \\ NoUnnecessaryIdling \\ NoPreemption \\ SuspendNonReadyProcs \end{array} \right]$$

4 Ada 95 implementation

The above model has a straightforward implementation using Burns and Wellings' method for non-preemptive scheduling in Ada 95 [3].

4.1 Global data

Global variables are used for communication between the application process and scheduler implementations. Each process is given a unique identifier, ranging from lowest to highest priority.

```
type ProcId is range 1.. $n$ ; --  $n$  application processes
```

An array holds the Ada task identifiers [5, §C.7.1] corresponding to each process. Volatile memory [5, §C.6] is used to safely implement variables shared between Ada tasks.

```
TaskIds : array(ProcId) of Task_ID;
pragma Volatile(TaskIds);
```

Another array holds the next arrival time [5, §D.8] for each periodic process.

```
Arrivals : array(ProcId) of Time;
pragma Volatile(Arrivals);
```

4.2 Shared variable implementations

Each of the application's shared variables v is simply declared as a global variable.

```
v : T_v; -- implements SharedVar_v
pragma Volatile(v);
```

4.3 Process implementations

Each periodic application process is implemented as an Ada task which initially suspends itself [5, §D.11], until released by the scheduler, then performs an invocation and suspends itself again, repeatedly.

```
task Process_i is
  pragma Priority(System.Default_Priority); -- high
end Process_i;

task body Process_i is
  Self : Task_ID := Current_Task; -- this task's name
begin
  TaskIds(i) := Self; -- associate Ada task id with 'i'
  Arrivals(i) := ...; -- beginning time (specification time '0')
  loop -- forever
    Hold(Self); -- suspend until next arrival
    -- application-specific code
    Arrivals(i) := Arrivals(i) + T_i; -- find next arrival time
  end loop;
end Process_i;
```

In this implementation S_{susp} is thus the time required to execute the `Hold` statement (assuming a process becomes 'not ready' as soon as this statement *begins*), and C_i is the time required to execute the remainder of the statements in the `loop`, including the overhead of iterating once. The beginning time initially assigned to `Arrivals` must be late enough that we know *all* task elaboration and initialisation activities will be completed by this time.

4.4 Scheduler implementation

The scheduler is implemented as a task which busy-waits until the arrival time of some application task, and then releases [5, §D.11] that task. It eval-

uates arrival times in array `Arrivals` in such an order as to give $Process_{i+1}$ priority over $Process_i$.

```

task Scheduler is
  pragma Priority(System.Default_Priority-1); -- low
end Scheduler;

task body Scheduler is
  Now : Time; -- current absolute time
begin
  loop -- forever
    Now := Clock; -- get time
    for Id in reverse ProcId loop -- check arrivals in pri. order
      if Arrivals(Id) <= Now then -- process is ready
        Continue(TaskIds(Id)); -- release process
        exit; -- to outer loop
      end if;
    end loop;
  end loop;
end Scheduler;

```

Thus S_{res} is the time required to execute the `Continue` statement, and S_i is the time required to execute sufficient iterations of the outer and inner loops to ensure that process i is selected. The worst case is when the clock read occurs *just before* i 's arrival time, and the arrival is not identified until the next *outer* iteration.

5 Schedulability testing

A disturbing feature of the process model in Section 3.2 (and its implementation in Section 4.3) is that it does not promise to complete each invocation of process i within the programmer's specified deadline D_i . This is because, *in isolation*, an individual process cannot know that it will receive sufficient computational resources to do so. Failure to receive sufficient run time in a period may cause a process invocation to miss its deadline, and even delay the arrival of subsequent invocations. The ability of each process to meet its deadlines is a property of the entire set of processes, and the scheduler, acting in concert!

An abstract specification of a real-time system will typically state that each process *must* perform a certain behaviour in every period, before its deadline expires. If we are to claim that the model in Section 3 is a valid refinement of such a specification, then we need some way of demonstrating that the model achieves this requirement. Fortunately, since the model in Section 3 follows established scheduling principles, the ability to do this can be proven by appealing to scheduling theory [1, 3]. The following test can be used to prove whether a system design expressed using the above components will always meet its deadlines or not.

Firstly, let C'_i be the worst-case execution time for an invocation of process i , including any scheduling overheads. This is the time for the scheduler to select i to run, plus the time for the scheduler to resume execution of i , plus the worst-case execution time of i , plus the time required to suspend i [3].

$$C'_i = S_i + S_{res} + C_i + S_{susp}$$

The schedulability test is then a straightforward variant of the well-known *response time* test [1]. It checks that, for all processes i , their worst-case response time R_i is less than their specified deadline D_i . The worst-case response time for an invocation of process i is calculated as the worst-case computation time of the current invocation, plus the time spent waiting for the process to be released. The process's worst-case *release time* r_i is the worst-case blocking time due to a single invocation of a lower-priority process k , plus the worst-case interference due to arrivals of higher-priority processes j during interval r_i (at least one for each higher-priority process) [3].

$$\forall i : I \bullet R_i \leq D_i$$

$$\text{where } R_i = r_i + C'_i$$

$$\text{and } r_i = \max_{k=1}^{i-1} C'_k + \sum_{j=i+1}^n \left(\left\lfloor \frac{r_i}{T_j} \right\rfloor + 1 \right) C'_j$$

6 Related work

Several earlier scheduling models are very similar to ours, but use different formalisms. Jackson [6] presented a model of real-time, non-preemptive scheduling in *CSP*, and used the FDR tool to analyse the ability of task sets expressed in this model to meet their deadlines. Liu et al [8] used the *Temporal Logic of Actions*, with ‘time’ introduced as one of the state variables

in state sequences, to model both preemptive and non-preemptive scheduling. Zhou et al [13, 7] used the trace-based *duration calculus* to express a scheduling model that included message-passing communication [13].

7 Conclusion

We have presented a formal model based on a recent proposal for implementing non-preemptive real-time multi-tasking [3]. Use of a trace-based specification notation [11] allowed concise definitions of quite complex behaviours.

Acknowledgements I wish to thank Alan Burns, Ian Hayes, Karl Lerner and Mark Utting for their many technical corrections to this work, and Peter Kearney for comments on drafts of this article. This work is funded by the Information Technology Division of the Defence Science and Technology Organisation.

References

- [1] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. Wellings. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, 8(5):284–292, September 1993.
- [2] I. J. Bate, A. Burns, and N. C. Audsley. Putting fixed priority scheduling theory into engineering practice for safety critical applications. In *Proc. Second Real-Time Applications Symposium*, pages 2–10, Boston, June 1996.
- [3] A. Burns and A. J. Wellings. Simple Ada 95 tasking models for high integrity applications. Department of Computer Science, University of York, May 1996.
- [4] K. Duddy, L. Everett, C. Millerchip, B. Mahony, and I. J. Hayes. Z-based notation for the specification of timing properties. Draft, Department of Computer Science, University of Queensland, June 1995.
- [5] ISO. *Ada Reference Manual: Language and Standard Libraries*, 6.0 edition, December 1994. International Standard ISO/IEC 8652:1995.
- [6] D. M. Jackson. Experiences in embedded scheduling. In M.-C. Gaudel and J. Woodcock, editors, *FME'96: Industrial Benefit and Advances in Formal Methods*, volume 1051 of *Lecture Notes in Computer Science*, pages 445–464. Springer-Verlag, 1996.

- [7] Z. Liu. Specification and verification in the duration calculus. In M. Joseph, editor, *Real-Time Systems—Specification, Verification and Analysis*, chapter 7, pages 182–228. Prentice-Hall, 1996.
- [8] Z. Liu, M. Joseph, and T. Janowski. Verification of schedulability for real-time programs. *Formal Aspects of Computing*, 7(5):510–532, 1995.
- [9] B. Mahony. Using the refinement calculus for dataflow processes. Technical Report 94-32, Software Verification Research Centre, October 1994.
- [10] B. Mahony. Networks of predicate transformers. Technical Report 95-5, Software Verification Research Centre, February 1995.
- [11] C. Millerchip, B. Mahony, and I. J. Hayes. The generic problem competition: A whole system specification of the boiler system. Software Verification Research Centre, University of Queensland, June 1993.
- [12] M. Utting and C. J. Fidge. A real-time refinement calculus that changes only time. In He Jifeng, John Cooke, and Peter Wallis, editors, *Seventh BCS-FACS Refinement Workshop*, Electronic Workshops in Computing. Springer-Verlag, 1996. <http://www.ewic.org.uk/ewic/>.
- [13] Zhou Chaochen, M. R. Hansen, A. P. Ravn, and H. Rischel. Duration specifications for shared processors. In J. Vytupil, editor, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 571 of *Lecture Notes in Computer Science*, pages 21–32. Springer-Verlag, 1992.