INCORPORATING UNBOUNDED ALGORITHMS INTO PREDICTABLE REAL-TIME SYSTEMS†

N. C. Audsley

A. Burns

M. F. Richardson

A. J. Wellings

Real-Time Systems Research Group,
Department of Computer Science,
University of York, UK

September 1991

ABSTRACT

The incorporation of unbounded components (i.e. software modules that cannot be analysed to produce realistic worst case execution times) into hard real-time applications has been recognised as a key issue for the next generation of systems. We present an architectural model that caters for the three main approaches to integrating unbounded components — imprecise computation, sieve functions and multiple versions. This architectural model is feasible because it is supported by schedulability tests that will guarantee the bounded tasks. These test are defined in the paper. A simple computational model that uses preemptive priority-based dispatching is required. The wide-spread use of techniques such as imprecise computation will only happen if they are integrated into standard software engineering methods. We therefore show how the techniques can be realised in Ada 9X.

† This work is supported, in part, by the UK Information Engineering Advanced Technology Programme, Grant GR/F 35920/4/1/1214

1. INTRODUCTION

The need to give 100% guarantees to hard real-time critical tasks is often seen to be at variance with the flexibility required to deal with non-deterministic environments and the incorporation of complex software algorithms. Indeed, in a recent editorial Stankovic and Ramamritham argue that[1]

> "*The next generation of real-time systems will be large, complex, distributed, adaptive, contain many types of timing constraints, need to operate in a highly non-deterministic environment, and evolve over a long system lifetime ... for these systems, the typical semantics (all tasks make their deadlines 100% of the time) associated with small static real-time systems is not sufficient.*"

However, crucial real-time tasks must still be guaranteed [2]. This has the consequence of:

- restricting the algorithms that can be used in these tasks (to those that can be analysed to produce non-pessimistic worst case execution estimates).
- the under usage of computing resources at run-time.

The under utilisation comes from:

(a) Software components not taking worst case execution paths.

(b) Hardware behaving better than estimated — due to gains from caching, pipelining or other facilities.

(c) Sporadic processes not executing at their maximum rate.

(d) Error handling software (e.g. exception handlers or alternatives in a recovery block model) not executing.

(e) Spare capacity that the schedulability analysis required in order to guarantee hard deadlines.

It is often stated that the soft real-time and non-real-time processes can utilise this spare capacity. Whilst this is undoubtedly true, we will investigate in this paper how the hard real-time processes themselves can have their utility (or value) improved at run-time. This will be done by incorporating unbounded software components into the architecture of the hard real-time activities. We use the term *unbounded* to imply that the timing behaviour of these components is unpredictable or is so pessimistic that no scheduling scheme will guarantee them.

Liu has recognised three ways by which unbounded components can be integrated into hard real-time systems (we assume a hard real-time system to consist of tasks that have deadlines specified relative to their release times) [3]:

- imprecise computations — for monotone tasks
- sieve functions — for non-monotone tasks
- multiple versions

Each of these models is described below:

1.1. Imprecise Computations

Monotone tasks are those that can produce intermediate results of value. Moreover, the value (or utility) of these results does not decrease as the execution of the task continues. With monotone tasks, it is usual to define a minimum precision that is needed by the application. The computations required to produce this minimum form the *mandatory* part of the task. Clearly this mandatory phase must have a worst case execution time (WCET) that is guaranteed by the schedulability analysis.

The rest of the task's execution is *optional*. If it executes it will improve the result but it is not necessary to guarantee any part of this optional phase; the mandatory part will ensure safe execution. The optional part is often structured as an iterating loop that progressively refines (and improves) the potential output of the task. It is not necessary to analyse the optional part, and therefore it may contain unbounded activities.

Computational algorithms that could contain monotone tasks, and hence benefit from imprecise computation, are found in many application domains, including numerical computation, statistical estimation and prediction, heuristic search activities, sorting, and database query processing [3].

1.2. Sieve Functions

Monotone tasks benefit from refinement after an adequate result has been computed. Tasks that can benefit from sieve functions utilise refinements during the production of any result. Consider a task to consist of a sequence of computational steps (i.e. a sieve); if all steps are compulsory then we have a standard task. However if one or more steps can be omitted (with an adequate result still being produced) then those steps can be implemented with unbounded components. Only the steps that must be executed need be analysed for WCET.

Sieve functions are appropriate when an old value can by either updated and used, to give a precise results, or used "as is" to produce a safe result.

1.3. Multiple Versions

The final model is the most widely applicable. Each system process is implemented as two distinct tasks. The *primary* version will produce a precise result but cannot be adequately analysed (i.e. it uses unbounded components). The *secondary* version is much simpler, can be analysed to give a WCET and will produce an adequate safe result. At run time the primary is allowed to execute first; if it produces a result then the secondary (predictable) version is never executed. However the secondary must always be guaranteed and hence at some predefined time the primary version has to be abandoned and the safe version run to completion.

1.4. Aims and Structure of Paper

All of the above models (and possible hybrids thereof) have a guaranteed adequate safe operational mode together with components that do not have to be guaranteed and hence do not have to be examined to determine their WCETs. It may however to useful to develop probabilistic arguments or measurements about the likelihood that, for example, the primary version will be run to completion in a multiple version implementation.

Imprecise computations have been studied at length by Liu *et al* [4, 5, 6, 7, 8, 9] in the context of a special purpose operating system *flex*. In this paper we show, in Section 2, how all three models can be supported by a single simple architecture based on preemptive (static) priority based scheduling. In addition we illustrate, in Section 3, how the resulting scheme can be implemented with a language such as Ada9X. For hard real-time systems, it is essential that any architectural model be supported by schedulability analysis. Consequently, in Section 4, we show how an offline schedulability test can be used to guarantee mandatory tasks. Having guaranteed the mandatory task, it is necessary to schedule the optional tasks such that overall effectiveness to the system is maximised. This

issue is discussed in Section 5. Finally, in Section 6 we present our conclusions.

2. BASIC ARCHITECTURAL MODEL

At run-time the dispatcher will deal with tasks (threads) that have been analysed for their temporal requirements and properties. We assume that each such task has the following structure:

Input, Computation, Output

i.e. it has an input phase followed by computational activity followed by the production of output. For time-triggered tasks the input phase may be missing.

If a task has an optional (unbounded) computational part, X, then the general model for a task becomes:

I, C1, X, C2, O

where C1 and C2 are phases of the required computation. These activities (plus I and O) will be analysed for their WCETs; X will not.

Each of the three models presented in Section 1 can be covered by this structure:

Imprecise Computation

With this model C2 is effectively null and C1 represents the mandatory part. The optional phase is represented by X.

Module C1 will produce an adequate result into a shared data area; X will read and write into this data area and O will then read this value. To protect against X only partially updating the result before O reads it, the data area must be accessed under mutual exclusion.

Sieve Functions

A task that has a number of optional parts can be split up into a number of precedence constrained tasks all of which have the structure outlined above (i.e. I, C1, X, C2, O). Here C1 and C2 both contain necessary code. A protected data object must be used for communication between the different parts.

Multiple Versions

Here the model is slightly different although the architecture is the same; C1 is absent (apart from any computation associated with the input action I). C2 has the form:

```
if X completed then
  do nothing
else
  execute secondary version
end if;
```

X implements the primary version. Note that if X has completed then C2 will finish early and further free (gain) time will be generated.

2.1. Computational Model

One of the motivations for this work is to use only simple (provable) run-time facilities. We therefore assume that each node in our (possible distributed) system has a fixed set of processes allocated. Each process will be assigned a distinct priority that will remain static (apart from when priority inheritance is required). The run-time dispatcher will implement a preemptive priority based mechanism.

2.2. Task Structures

Each hard real-time process, with deadline D, is (from a scheduling standpoint) decomposed into three tasks representing the *prologue* (τ^p), *epilogue* (τ^e) and *optional* (τ^o) parts:

τ^p = {I, C1}
τ^e = {C2, O}
τ^o = {X}

Consider periodic tasks. The two bounded tasks, τ^p and τ^e have equal period but their releases are out of phase by virtue of *offsets*. The offset of a task is the time at which the first release of that task occurs. Subsequent releases occur at intervals defined by the task period. The following points apply:

- τ^p and τ^e have the same period.
- τ^p has an offset of 0 and a deadline of S, ($S < D$).
- τ^e has an offset of S and a deadline of $D - S$.
- τ^o can only do effective work between the completion of τ^p and S.

Let us assume that this set of tasks can be scheduled. Therefore τ^p completes before τ^e starts with data passing from τ^p to τ^e via a data object. If τ^p completes before the release of τ^e at S, τ^o may execute. When τ^e is released, it will preempt τ^o if τ^o were executing (as the priority of τ^e is greater than that of τ^o). Task τ^e now executes.

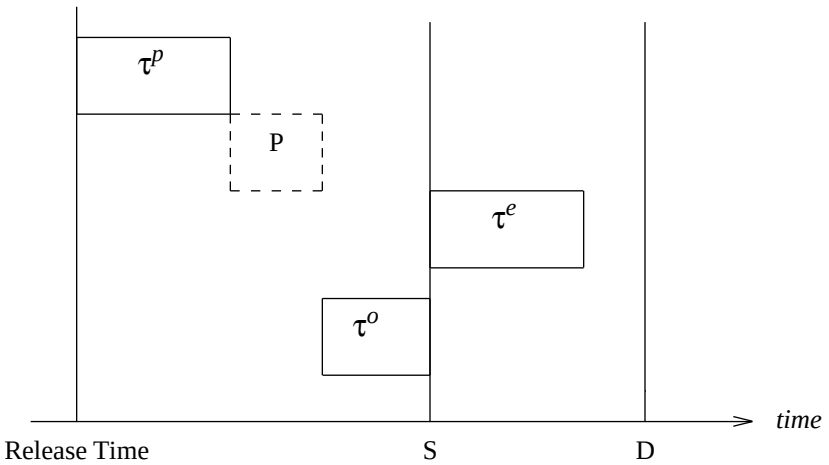

Figure 1

An example relationship between τ^p, τ^e and τ^o is shown is Figure 1. Here τ^p and τ^e execute as soon as they are released (i.e. there are no higher priority processes runnable); τ^o has some processor time before τ^e preempts it, however it does not run immediately τ^p completes since some other high priority process (P) executes first. The choice of S is

arbitrary whilst maintaining a schedulable system. The value of S can be modified as part of the iterative system design process, to maximise the time for τ^o to execute.

Sporadic tasks can be considered in the same manner. Here, the minimum inter-arrival times of τ^p and τ^e are identical. We assume that the latter is released *S* units of time after τ^p.

Preemptive static priority based scheduling is used and hence priorities must be assigned to all the prologue, epilogue and optional tasks. We shall represent the tasks sets by $[\tau^p]$, $[\tau^e]$ and $[\tau^o]$. The schedulability analysis will assign priorities to $[\tau^p]$ and $[\tau^e]$ so that they are guaranteed to meet their deadlines (see section 4.1); $[\tau^o]$ are given base priorities below those of $[\tau^p]$ and $[\tau^e]$ so that they have the maximum effect on the system's overall performance (see section 5). There are however two ways in which $[\tau^o]$ can be implemented.

Simple Model

If at any time between the completion of τ^p and the start of τ^e the processor has no other prologue or epilogue task to run then a τ^o can execute. The simple model will therefore allow a τ^o to execute in time that is free in the schedule. It will also soak up time gained from other tasks (including τ^p) executing for less than their WCET (this is termed *gain time*).

When there are no $[\tau^p]$ or $[\tau^e]$ to execute then the τ^o with the highest priority will run.

Inheritance Model

There are situations that can arrive in the above model that will not allow τ^o to benefit from gain time. To illustrate this, consider τ^p with a sizable computational requirement that completes early yielding a gain time of G. Now G is only valid until time S (i.e. when τ^p should have finished. As G was guaranteed (when it was part of τ^p) then any task can use it (in particular τ^o could use it). However there may be a lower priority (real-time) task (Q1 say) that is executable in the interval up to S. Because of τ^p finishing early Q1 will execute early thereby postponing the execution of τ^o. It may be that Q1 will not complete until after S hence the free time (G) will become free only after S, hence τ^o cannot utilise it. To improve on this situation, τ^o could inherit the priority of τ^p for the duration of G; it would then run immediately.

Section 3 illustrates how the above two models can be programmed.

2.3. Implication for Task Allocation in a Distributed System

The model presented above transforms one logical task into potentially three tasks for execution at run-time. Our approach assumes that these three tasks are tightly coupled, and therefore should be executed on the same processor in a distributed system. Consequently, there is an obligation upon any task/processor allocation algorithm to recognise this.

3. ADA 9X REPRESENTATION

In this section we consider how the architectural model presented in Section 2 can be represented in Ada 9X. We do this to illustrate how a standard language, albeit the revised version of Ada, can be used to implement all three models. The use of Ada 83 is more

problematic although Liu has given some implementation guidelines[10].

Given the current status of the Ada 9X project it is not possible to know for certain what the actual syntax and semantics of any new facilities will be. However, there is now a proposed Ada 9X mapping document [11, 12] so we shall use the syntax and semantics presented there. In particular we shall assume the following.

1) Ada 9X will provide a fast mutual exclusion primitive. This will take the form of a protected record.
2) Ada 9X will provide a mechanism for asynchronous transfer of control. This is an extension to the select statement allowing an "in" clause.
3) Ada 9X will provide a mechanism to give an accurate representation of a periodic task; in particular the delay statement is overloaded to take a calendar-time parameter.
4) Ada 9X will provide a dynamic priority facility.

Details of the new language features are given in Appendix 1.

3.1. Representation of the Simple Model

We assume that τ^p on release computes the result of the prologue and then places the data in a protected record. It then delays until its next release. Consider the following protected record:

```
protected SHARED_DATA is

    pragma PRIORITY(MAX);  -- MAX is set higher than P1 or P2
    procedure WRITE(D : DATA);
    entry READ(D: out DATA; DATA_STILL_AVAILABLE : in BOOLEAN);

record
    THE_DATA : DATA;
    DATA_AVAILABLE : BOOLEAN := FALSE;
end SHARED_DATA;

protected body SHARED_DATA is

    procedure WRITE(D : DATA) is
    begin
        THE_DATA := D;
        DATA_AVAILABLE := TRUE; -- indicate that the
                                -- data is available
    end WRITE;

    entry READ(D : out DATA; DATA_STILL_AVAILABLE : in BOOLEAN)
              when DATA_AVAILABLE is
    begin
        D := THE_DATA;
        DATA_AVAILABLE := DATA_STILL_AVAILABLE;
    end READ;
end SHARED_DATA;
```

It has two operators, one to allow data to be placed in it and the other to enable the data to

be read. τ^p can therefore be implemented as:

```
task T_PROLOGUE is
     pragma PRIORITY(P1);
end T_PROLOGUE;

task body T_PROLOGUE is
   -- declaration of RESULT and TO_NEXT_PERIOD etc
begin

   loop
        -- compute RESULT
        SHARED_DATA.WRITE(RESULT);
        delay(TO_NEXT_PERIOD);
   end loop;
end T_PROLOGUE;
```

To implement τ^o requires that it reads the latest result from τ^p and then performs the optional computation replacing the results in the shared data at the end. However, τ^o must be interruptible during this computation by τ^e. If it is interrupted then it returns to wait for more data. τ^o can be implemented as:

```
task T_OPTIONAL is
     pragma PRIORITY(P2);  -- P2 less than P1 or P3
end T_OPTIONAL;

task body T_OPTIONAL is
   -- declaration of VALUE, UPDATE etc
   DATA_STILL_AVAILABLE : constant BOOLEAN := TRUE;
begin
   loop
        select
             T_EPILOGUE.OUT_OF_TIME;
        in
            SHARED_DATA.READ(VALUE, DATA_STILL_AVAILABLE);
            -- compute optional RESULT
            -- potentially this may contain
            -- loop
            --     NEXT_INTERACTION;
            --     exit WHEN_BEST_RESULT_OBTAINED;
            --     SHARED_DATA.WRITE(UPDATE);
            -- end loop;
            SHARED_DATA.WRITE(UPDATE);
        end select
   end loop;
end T_OPTIONAL;
```

Task τ^e can be implemented thus:

```
task T_EPILOGUE is
     pragma PRIORITY(P3);
     entry OUT_OF_TIME;
end T_EPILOGUE;

task body T_EPILOGUE is
   -- declaration of VALUE,OFFSET etc
   DATA_UNAVAILABLE : constant BOOLEAN := FALSE;
begin
   delay(OFFSET);
   loop
        select
            accept OUT_OF_TIME; -- stop T_OPTIONAL
        else
            null;
        end select;
        SHARED_DATA.READ(VALUE, DATA_UNAVAILABLE);
        -- compute final RESULT
        delay(TO_NEXT_PERIOD);
   end loop;
end T_EPILOGUE;
```

When τ^e starts to execute it will accept the call (if it has been made) from τ^o on `OUT_OF_TIME`. When τ^o next executes it will break out of the select statement and return, via the loop, to reading from the protected record again.

The solution outlined above is suitable for both imprecise computations and sieve functions. It can easily be extended to multiple versions by having the data structure contain a flag to indicate that the primary version has completed and therefore the secondary need not run.

The implementation of the inheritance model requires added support from the underlying run-time operating system. This is described in Appendix 2.

4. SCHEDULABILITY TESTS AND PRIORITY ASSIGNMENT

The unbounded computation model presented in section 2 requires that all prologue and epilogue task deadlines are guaranteed. This guarantee is provided by a schedulability test which takes the timing characteristics of the tasks and indicates whether those deadlines will be met at runtime (if appropriate priority assignments are made). The unbounded model raises difficult schedulability problems. Indeed, in general, the problem of optimally scheduling a set of tasks (some with offsets) is known to be NP-hard [13]. We now present a sub-optimal method for deciding whether deadlines will be met. We assume a preemptive (static) priority based scheduling approach.

Determining whether task deadlines can be guaranteed in a static priority system involves two distinct stages:

- assign distinct priorities to each task;
- use a schedulability test to analyse this priority assignment.

By the unbounded computation model (section 2), all tasks are characterised as having either known periods (for cyclic tasks) or known worst case arrival rates (for sporadic tasks)[†]. Each task also has a deadline which is, in general, less than its period. We assume

that all bounded tasks have been analysed to obtain their worst case execution times. The optional tasks are ignored from a schedulability viewpoint as their deadlines are not guaranteed.

The following section defines all task timing characteristics. Subsequent sections address the issues of priority assignment and schedulability.

4.1. Task Timing Characteristics

The set of all processes, Λ, has cardinality m, with individual processes labelled $\Lambda^1, \Lambda^2, \ldots, \Lambda^m$. Each Λ^i has a deadline of D^i and a period of T^i. Each process Λ^i is decomposed into prologue, epilogue and optional tasks τ^{p_i}, τ^{e_i} and τ^{o_i} respectively, although the latter is ignored for schedulability purposes. The intermediate deadline is termed S^i (and is relative to the release time).

The period of each task, T^{p_i} and T^{e_i}, is equal to T^i. The computational requirements of each task are C^{p_i} and C^{e_i}. The offsets and deadlines of the tasks are:

$$O^{p_i} = 0$$

$$O^{e_i} = S^i$$

$$D^{p_i} = S^i$$

$$D^{e_i} = D^i - S^i$$

Initially, we select an arbitrary S^i to divide equally the slack time between the τ^{p_i} and τ^{e_i}. Thus:

$$S^i = \frac{D^i - (C^{p_i} + C^{e_i})}{2} + C^{p_i}$$

Successive releases of a task, for example τ^{p_i} occur at O^{p_i}, and then at $O^{p_i} + T^{p_i}$, $O^{p_i} + 2T^{p_i}$ etc.

4.2. Priority Assignment

The familiar rate monotonic assignment model cannot be used with the above task set as one of the requirements for the rate monotonic model, period equals deadline, is not satisfied. Rather the deadline monotonic assignment method must be used [13, 14]. With this method static priorities are assigned according to deadlines. The shorter the deadline, the higher the priority.

When tasks are permitted to have arbitrary offsets, deadline-monotonic priority assignment ceases to be optimal. However, since no optimal priority assignment has appeared in the literature, we assume deadline-monotonic priority assignment.

To denote the priority of a task, we introduce a subscript. Thus, $\tau_j^{p_i}$ has an assigned priority of j. Priorities are arranged (following convention) such that 1 is the highest, with n the lowest (there being n tasks to which priorities are assigned, where $n = 3m$).

For the purposes of the following section, a shorthand is introduced. We refer to the

---

† From a schedulability point of view each sporadic task is modelled as a cyclic task whose period is derived from its worst case arrival rate.

task at priority level j (for example $\tau_j^{p_k}$) as τ_j. The timing characteristics of $\tau_j^{p_k}$, that is O^{p_k}, D^{p_k}, T^{p_k} and C^{p_k}, are referred to as O_j, D_j, T_j and C_j respectively.

4.3. Schedulability Tests

A number of schedulability tests have been produced for the deadline monotonic priority ordering [13, 15, 16, 14]. Most have concentrated on sufficient and necessary conditions. In the general case, the problem is NP-hard implying that such tests are exponential in complexity. In [16, 14], the authors develop a simple sufficient and not necessary test of $O(n^2)$ complexity which we now develop to provide a schedulability test for the unbounded computation task model.

The simple test takes the following form: for all tasks, the following must hold

$$C_i + I_i + B_i \leq D_i \tag{1}$$

where

$$I_i = \sum_{j=1}^{i-1} \left\lceil \frac{D_i}{T_j} \right\rceil C_j$$

The interference value, I_i, is the maximum time that higher priority tasks can run in the interval defined by the release and deadline of τ_i. This is at its maximum value when all tasks are released at the same time — it is called the *critical instant* [17]. This instant represents the worst-case loading on the processor. If a process can meet its deadline for a release beginning at a critical instant it will always meet its deadlines.

The blocking factor, B_i, relates to the total time, in the interval of interest, that a lower priority process is executing. With a preemptive dispatcher it would appear that this value should be zero. However, process interactions inevitably give rise to *priority inversion* [18]. For example, after τ^{p_i} has completed, τ^{o_i} can execute. This process may be executing inside the protected object providing mutual exclusion when τ^{e_i} is released, where the latter has a higher priority than τ^{o_i}. To bound the time that τ^{e_i} must wait (for τ^{o_i} to complete) some form of *priority inheritance* is implemented. With the Ada 9X model we saw that the use of immediate priority ceiling inheritance gives an adequate method of bounding priority inversion. Therefore, the blocking factor, B_k, for any process τ_k, is bounded to be the maximum time a lower priority process will take in executing within a protected object that has a higher ceiling priority (than k).

The test formulated above assumes that processes do not have offsets. This implies that a critical instant occurs when the system is initiated (time 0) and then at intervals defined by the least common multiple of all process periods. When considering the form of process pairs within the unbounded model, this estimation of interference is pessimistic: no τ^{p_i} τ^{e_i} pair execute simultaneously. Whilst the above test could be used, a better schedulability test is now derived.

Consider task τ_i. The only tasks that can prevent it meeting its deadline are those of a higher priority. When task pair (τ^{p_j} τ^{e_j}) both have higher priority, we assume that τ_i and the task in the pair with the greatest computation time are released simultaneously at time 0. The task with the lower computation time of the pair is initially released at an offset from 0 (this offset is defined in the following discussion). When only one of a pair has higher priority than τ_i, we assume that it is released simultaneously with τ_i. This scenario represents the maximum interference by the tasks of higher priority on τ_i†.

† a proof of this is easily provided by extending the proof for theorem 1 in [17].

For schedulability purposes only, we transform the offsets of pairs of processes if the epilogue task has longer computation time than the prologue. That is, for each τ^{p_i} τ^{e_i} pair (where $C^{p_i} < {}^{e}C_i$) we reassign the offsets of the task pair:

$$O^{p_i} = T^i - S^i$$

$$O^{e_i} = 0$$

We can then restate equation (1) to provide a schedulability test:

$$C_i + I_i + B_i \leq D_i \qquad (2)$$

where

$$I_i = \sum_{j=1}^{i-1} \left\lceil \frac{D_i - O_j}{T_j} \right\rceil C_j$$

Note that whilst calculating the interference on τ_i, we assume the computation time of its partner task is temporarily set to 0.

Whilst an obvious improvement on equation (1), this schedulability test remains non optimal. This is due to considering the effect of higher priority processes on τ_i individually. It is clear that the effects of interleaving may cause some executions to occur after D_i, therefore beyond interference considerations. However, the test presented above is sufficient, as the worst possible interference is calculated. A sufficient and necessary can be derived from the sufficient and not necessary schedulability test given in Audsley *et al* [14] in the same way as we derived the sufficient test in equation (2).

4.4. Summary

In this section we have introduced a priority assignment and schedulability test for our process model.

The schedulability test presented is sufficient and not necessary. The complexity is $O(n^2)$ in the number of processes. As indicated, a pseudo polynomial sufficient and necessary test is easily derived, although the issue of defining an optimal priority assignment for offset process sets detracts from its current usefulness.

The schedulability test can easily be incorporated into an appropriate CASE tool. Such a tool can incorporate

(a) the modification of S values;

(b) assignment of priorities;

(c) checking of schedulability.

5. MAXIMISING SYSTEM UTILITY

The assignment of priority values to the τ^o tasks although restricted to values below $[\tau^p]$ and $[\tau^e]$ is nevertheless important if the tasks are to contribute to increasing the utility of the system. Some systems have experimented with the use of *time-value* scheduling (only) to determine the complete ordering of executions [19]. We are unaware of anyone who has combined preemptive priority based scheduling (using time attributes such as deadline or period) with a time-value scheduling of non-crucial tasks.

It would clearly be possible to use a complex dynamic decision process to decide at

any one time which of the available τ^o tasks to run. Such algorithms require further study; in this paper we present a straightforward approach which requires no dynamic decision making. All [τ^o] are given priorities that are then used by the preemptive priority-based dispatcher.

In his thesis on time-value scheduling Locke presents the following result — which we use [20]:

> "*Given a set of processes with precisely known constant values for completing them, it can be shown that a sequence of processes in decreasing order by value density (V/C where V is its value and C is its processing time) will produce a total value at every process completion time least as high as any other schedule.*"

To compare [τ^o] there must be a system-wide single measure of value, or to use a less confusing term, *utility*. Each τ^o must be given a utility value that is a estimation of the benefit to the entire system of its execution. If possible, those estimations must be given as a value per unit time of execution.

Once utility values have been derived, they are mapped on to execution priority levels; the higher the utility, the higher the priority.

The above method of dealing with unbounded tasks has the advantage of simplicity but does not cater for all situations. In particular, a τ^o task implementing a primary version may be preempted by another optional task just before it completes. It may not execute again before its associated τ^e must execute. This τ^e task may then preempt the other τ^o tasks so that none of them produce a worthwhile result. In effect the utility (per unit time) of the primary version increases as it nears its completion. This could be accommodated by allowing it to raise its priority as it progresses (within the range for τ^o tasks).

To assign a single utility value to a task that iterates towards an improved solution is also too simplistic as its first iteration may produce a dramatic effect (i.e. high utility) but subsequent iterations may have only a marginal effect (i.e. low utility). These τ^o tasks must be allowed to suspend themselves if they know they can do no further useful work. Ideally, they should be able to lower their priority to reflect their lower utility.

6. CONCLUSION

The incorporation of unbounded components into hard real-time applications has been recognised as a key issue for the next generation of systems. We have presented an architectural model that caters for the three main ways of integrating unbounded components — imprecise computation, sieve functions and multiple versions. This architectural model is feasible because it is supported by schedulability tests that will guarantee (or not) the bounded tasks. A simple computational model that uses preemptive priority-based dispatching is required. The complexity of the approach is thus focused on the pre-run-time analysis, not the run-time scheduling.

The wide-spread use of techniques such as imprecise computation will only happen if they are integrated into standard software engineering methods. We have therefore shown how the techniques can be realised in Ada 9X.

Continuing work in this area focuses on two issues:

- The use of global search methods such as simulated annealing to automate the choice of the S values.
- The use of more dynamic methods of ordering the executions of the τ^o tasks.

APPENDIX 1: Ada 9X Language Features

In this appendix we briefly summarise those features of Ada 9X that are relevant to the implementation model presented in this paper.

Protected Records

A protected record in Ada 9X is a mechanism which provides fast mutual exclusion. It is similar in concept to a conditional critical region [21] and a monitor [22]. Data which is to be accessed in mutual exclusion is encapsulated in a protected record. This data can only be accessed by procedures and entries also declared in the protected record. Execution of a procedure or an entry requires exclusive access to the record. As with tasks in Ada, protected records may be declared as instances of a protected record type, or a single one-off protected record (of anonymous type). For example, the following protected record is a single instance of an anonymous protected record type which allows mutually exclusive access to shared data:

```
protected SHARED_DATA is

    pragma PRIORITY(P);
    -- operations on the shared data
    procedure WRITE(D : DATA);
    procedure READ(D: out DATA);

record
    THE_DATA : DATA := SOME_INITIAL_VALUE; -- the encapsulated data
end SHARED_DATA;

protected body SHARED_DATA is

    procedure WRITE(D : DATA) is
    begin
        THE_DATA := D;
    end WRITE;

    procedure READ(D : out DATA) is
    begin
        D := THE_DATA;
    end READ;
end SHARED_DATA;
```

The difference between a procedure and an entry in a protected record is as follows. A procedure simply provides mutually exclusive access to the data. If there is no other task active in the protected record then a call on the procedure will gain immediate access to the data. An entry may have an associated guard. A call to a guarded entry will only be allowed when the guard evaluates to true and there is no other process active in the protected record. Tasks which call entries that are closed are blocked (unless they made conditional entry calls) until the guard is true. The following shows the above reader writer example when a write must initialise the data before it is read.

```
protected SHARED_DATA is

    pragma PRIORITY(P);
    procedure WRITE(D : DATA);
    entry READ(D: out DATA);

record
    THE_DATA : DATA;
    DATA_AVAILABLE : BOOLEAN := FALSE;
end SHARED_DATA;

protected body SHARED_DATA is

    procedure WRITE(D : DATA) is
    begin
        THE_DATA := D;
        DATA_AVAILABLE := TRUE; -- indicate that the
                                -- data is available
    end WRITE;

    entry READ(D : out DATA) when DATA_AVAILABLE is
    begin
        D := THE_DATA;
    end READ;
end SHARED_DATA;
```

On a single processor system, mutual exclusion of a protected record is implemented using priority. Each protected record has an associated priority. This priority is set to at least the highest priority of all the tasks which call its procedures and entries. As Ada has preemptive priority based scheduling, once a task which wishes to access a protected record is executing, no other task can be active inside the protected record. This priority is called the *ceiling* priority of the protected record.

To present the full semantics of protected records is beyond the scope of this paper. See the Ada 9X mapping documents for a full description [11, 12].

Asynchronous Transfer of Control

Ada 9X provides an asynchronous transfer of control facility by extending the Ada 83 select statement. Originally in Ada 83 a task making an entry call could only call one entry at once but could place a timeout on the acceptance of the call (or make a conditional entry call). In Ada 9X this has been generalised into a multiway select statement where a task can wait for one or more task entry calls (or protected record entry calls); again these may be conditional or have timeouts. Hence the following is legal Ada 9X.

```
-- wait for entry E1 in task T1,
-- or E2 in protected record P2, to be accepted
select
    T1.E1;
or
    P2.E2;
or
    delay 10.0;
end select;
```

Asynchronous transfer of control is provided by adding an *in* clause to the multiway select. For example the following select statement will execute the sequence of statements associated with the *in* clause if the call to `ERROR_MONITOR.ERROR_OCCURRED` cannot be accepted immediately. If, during the execution of the sequence of statements, the call can be accepted then it is accepted; the sequence of statements is abandoned and control is transferred to the error handling code following the entry call.

```
select
    ERROR_MONITOR.ERROR_OCCURRED;
    -- error handling code
in
    -- sequence of statement
end select;
```

This facility can also be used with a timeout to place deadlines on arbitrary sections of code. For example:

```
-- time critical section
select
    delay 5.0;
in
    -- code which must be executed with 5 seconds
end select;
```

Periodic Tasks

Although the Ada language has no direct representation for periodic tasks, it provides the delay statement with which a periodic task structure can be constructed. It is important to appreciate that "delay" is an approximate time construct, the inclusion of which in a task indicates that the task will be delayed by at least the amount defined. The following code illustrates a periodic task in Ada 83.

```
declare
  NEXT : TIME;
  INTERVAL : constant DURATION := ...;
begin
  NEXT := CLOCK + INTERVAL;
  loop
    -- code to be executed
    delay NEXT - CLOCK;
    NEXT := NEXT + INTERVAL;
  end loop;
end;
```

Unfortunately, the construct is not very accurate as the task may be preempted whilst executing the delay statement itself (i.e. after evaluating `NEXT - CLOCK`, but before posting the delay). It is for this reason that Ada 9X will allow the delay statement to take a time of day parameter.

Dynamic Priorities

Dynamic priorities are supported in Ada 9X by the following package.

```
with SYSTEM;
package DYNAMIC_PRIORITY_SUPPORT is

    subtype ANY_PRIORITY is INTEGER range
      SYSTEM.PRIORITY'FIRST .. SYSTEM.INTERRUPT_PRIORITY'LAST;

    procedure SET_PRIORITY(T : SYSTEM.TASK_CLASS,
                           PRIORITY : ANY_PRIORITY);
    procedure SET_PRIORITY(PRIORITY : ANY_PRIORITY);

    ....

end DYNAMIC_PRIORITY_SUPPORT;
```

It should be noted that Ada 9X has many more object-oriented programming facilities than Ada 83. In particular the derived type mechanism has been extended to provide an inheritance mechanism. A result of this is that all tasks are now derived from a common system class called `SYSTEM.TASK_CLASS`. An object of type `SYSTEM.TASK_CLASS` can refer to any task in the system.

APPENDIX 2: Representation of the Inheritance Model in Ada 9X

Representing the inheritance model is difficult to express directly in Ada 9X (or any other current real-time programming language). It requires that an application be able to ascertain how much CPU time (budget time) it has consumed during its execution so that it can calculate how much CPU time has been gained (by completing early). This requires help from the underlying run-time support systems. Traditionally extensions to the Ada run-time environment have been defined using Ada packages which interface directly into the run-time support systems (for example the ARTEWG CIFO [23]). It is therefore possible to imagine the following packages [24].

```
with CALENDAR; use CALENDAR;
package CPU_BUDGETING is

    type BUDGET_ID is private;
    type CPU_DURATION is <IMPLEMENTATION-DEFINED>;
    NULL_ID : constant BUDGET_ID;
    BUDGET_OVERRUN  : exception;
                      -- raised when budget time exceeded
    function SET_BUDGET(LIMIT : CPU_DURATION) return BUDGET_ID;
    function TIME_LEFT return CPU_DURATION;

    procedure CANCEL(B : in out BUDGET_ID); -- B is set to NULL_ID
private
    type BUDGET_ID is new INTEGER;
    NULL_ID : constant BUDGET_ID := 0;
end CPU_BUDGETING;
```

The above package enables a task to inform the run-time environment of the amount of CPU time that has been allocated to it. It also allows a task to enquire how much CPU time it has left from its budget. (The package provides other facilities for raising an exception if the budget time is exceeded and for cancelling the budget).

Once the "gain" time has been established a further interface into the run-time is required to change the priority of the optional task, τ^o, after it has consumed a certain amount of CPU time. This facility is illustrated below.

```
-- set the priority of the calling task to P
-- after it has consumed B execution time
procedure SET_PRI_AFTER_BUDGET(P: PRIORITY; B: DURATION_TYPE);
```

Only the outstanding call to the procedure is allowed; subsequent calls override previous calls. The inheritance model can now be illustrated. First the SHARED_DATA is extended to include the remaining CPU time:

```
protected SHARED_DATA is

    pragma PRIORITY(MAX);
    procedure WRITE(D : DATA; T : DURATION);
    entry READ(D: out DATA; DATA_STILL_AVAILABLE : in BOOLEAN;
               T : out DURATION );

record
    THE_DATA : DATA;
    TIME_LEFT : DURATION;
    DATA_AVAILABLE : BOOLEAN := FALSE;
end SHARED_DATA;

protected body SHARED_DATA is

    procedure WRITE(D : DATA; T : DURATION) is
    begin
        THE_DATA := D;
        TIME_LEFT := DURATION;
        DATA_AVAILABLE := TRUE; -- indicate that the data is available
    end WRITE;

    entry READ(D : out DATA; DATA_STILL_AVAILABLE : in BOOLEAN;
               T : out DURATION )
               when DATA_AVAILABLE is
    begin
        D := THE_DATA;
        T := TIME_LEFT;
        DATA_AVAILABLE := DATA_STILL_AVAILABLE;
    end READ;
end SHARED_DATA;
```

Next the prologue measures its execution time (O is some small value representing the execution time consumed in the remainder of the task). It then sets the priority of the optional task to P1. (We assume that the time left is sufficiently large to make the inheritance worthwhile.)

```
task T_PROLOGUE is
     pragma PRIORITY(P1);
end T_PROLOGUE;

task body T_PROLOGUE is
   -- declaration of RESULT and TO_NEXT_PERIOD etc
begin

   loop
        ID := CPU_BUDGETING.SET_BUDGET(..);
        -- compute RESULT
        SHARED_DATA.WRITE(RESULT, CPU_BUDGETING.TIME_LEFT - O);
        DYNAMIC_PRIORITY_SUPPORT.SET_PRIORITY(T_OPTIONAL, P1)
        CPU_BUDGETING.CANCEL_BUDGET(ID);
        delay(TO_NEXT_PERIOD);
   end loop;
end T_PROLOGUE;
```

The optional task now initially runs at the same priority as the prologue task. As soon as it executes it informs the run-time that its priority should be lowered after it has consumed the inherited budget time. It then executes the optional code. If it completes this code, it lowers its own priority in case it hasn't consumed all the budget time.

```
task T_OPTIONAL is
     pragma PRIORITY(P2);
end T_OPTIONAL;

task body T_OPTIONAL is
   -- declaration of VALUE, UPDATE etc
   DATA_STILL_AVAILABLE : constant BOOLEAN := TRUE;
begin
   loop
        select
             T_EPILOGUE.OUT_OF_TIME;
        in
            SHARED_DATA.READ(VALUE, DATA_STILL_AVAILABLE, TIME_LEFT));
            SET_PRI_AFTER_BUDGET(P2, TIME_LEFT);
            -- compute optional RESULT
            -- potentially this may contain
            -- loop
            --     NEXT_INTERACTION;
            --     SHARED_DATA.WRITE(UPDATE);
            --     exit WHEN_BEST_RESULT_OBTAINED;
            -- end loop;
            SHARED_DATA.WRITE(UPDATE);
            DYNAMIC_PRIORITY_SUPPORT.SET_PRIORITY(P2);
        end select;
   end loop;
end T_OPTIONAL;
```

The implementation of Task τ^e is the same as before.

```
task T_EPILOGUE is
     pragma PRIORITY(P3);
     entry OUT_OF_TIME;
end T_EPILOGUE;

task body T_EPILOGUE is
   -- declaration of VALUE,OFFSET etc
   DATA_UNAVAILABLE : constant BOOLEAN := FALSE;
begin
   delay(OFFSET);
   loop
        select
            accept OUT_OF_TIME;
        else
            null;
        end select;
        SHARED_DATA.READ(VALUE, DATA_UNAVAILABLE);
        -- compute final RESULT
        delay(UNTIL_NEXT_PERIOD);
   end loop;
end T_EPILOGUE;
```

References

1. J. A. Stankovic and K. Ramamritham, ‘‘What is Predictability for Real-Time Systems?’’, *Real-Time Systems* **2**(4), pp. 247-254 (1990).
2. A. Burns and A. J. Wellings, ‘‘Criticality and Utility in the Next Generation’’, *Real-time Systems* **3**(4) (1991).
3. J. W. S. Liu, K. J. Lin, W. K. Shih, A. C. S. Yu, J. Y. Chung and W. Zhao, ‘‘Algorithms for Scheduling Imprecise Computations’’, *IEEE Copmuter*, pp. 58-68 (May 1991).
4. J. W. S. Liu, K. J. Lin and S. Natarajan, ‘‘Scheduling Real-Time, Periodic Jobs Using Imprecise Results’’, *Proceedings 8th IEEE Real-Time Systems Symposium*, Fairmont Hotel, San Jose, California, pp. 252-260 (1-3 December 1987).
5. W. K. Shih, J. W. S. Liu , J. Y. Chung and D. W. Gillies, ‘‘Scheduling Tasks with Ready Times and Deadlines to Minimise Average Error’’, *Operating Systems Review* **23**(3), pp. 14-28 (July 1988).
6. J. Y. Chung, W. K. Shih, J. W. S. Liu and D. W. Gillies, ‘‘Scheduling imprecise computations to minimize total error’’, *Microprocessing & Microprogramming* **27**(1-5), pp. 67-74, 15th IEEE EUROMICRO Symposium on Microprocessing and Microprogramming (August 1989).
7. W. K. Shih, J. W. S. Liu and J. Y. Chung, ‘‘Algorithms for Scheduling Imprecise Computations with Timing Constraints’’, *Proceedings 10th IEEE Real-Time Systems Symposium*, Santa Monica, California (5-7 December 1989).
8. X. Song and J. W. S. Liu, ‘‘Performance of Multiversion Concurrency Control Algorithms in Maintaining Temporal Consistency’’, Dept. of Computer Science, Univ. of Illinois at Urbana-Champaign (February 1990).
9. J. Y. Chung, J. W. S. Liu and K.J. Lin, ‘‘Scheduling Periodic Jobs That Allow

Imprecise Results'', *IEEE Transactions on Computers* **39**(9), pp. 1156-1174 (September 1990).

10. J. W. S. Liu and K. J. Lin, ''On Means to Provide Flexibility in Scheduling'', *ACM Ada Letters* **8**(7), International Workshop on Ada Issues, Mortenhampstead, Devon, UK, 1-3 June 1988 (Fall 1988).
11. Intermetrics, ''Draft Mapping Document'', Ada 9X Project Report (February 1991).
12. Intermetrics, ''Draft Mapping Rationale Document'', Ada 9X Project Report (February 1991).
13. J. Y. T. Leung and J. Whitehead, ''On the Complexity of Fixed-Priority Scheduling of Periodic, Real-Time Tasks'', *Performance Evaluation (Netherlands)* **2**(4), pp. 237-250 (December 1982).
14. N. C. Audsley, A. Burns, M. F. Richardson and A. J. Wellings, ''Hard Real-Time Scheduling: The Deadline Monotonic Approach'', *Proceedings 8th IEEE Workshop on Real-Time Operating Systems and Software*, Atlanta, GA, USA (15-17 May 1991).
15. J. P. Lehoczky, ''Fixed Priority Scheduling of Periodic Task Sets With Arbitrary Deadlines'', *Proceedings 11th IEEE Real-Time Systems Symposium*, Lake Buena Vista, FL, USA, pp. 201-209 (5-7 Decmeber 1990).
16. N. C. Audsley, ''Deadline Monotonic Scheduling'', YCS 146, Department of Computer Science, University of York (October 1990).
17. C. L. Liu and J. W. Layland, ''Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment'', *Journal of the ACM* **20**(1), pp. 40-61 (1973).
18. L. Sha, R. Rajkumar and J. P. Lehoczky, ''Priority Inheritance Protocols: An Approach to Real-Time Synchronisation'', *IEEE Transactions on Computers* **39**(9), pp. 1175-1185 (September 1990).
19. J. D. Northcutt, *Mechanisms for Reliable Distributed Real-tine Operating Systems: The Alpha Kernel*, Academic Press, Orlando (1987).
20. C. D. Locke, ''Best-Effort Decision Making for Real-Time Scheduling'', CMU-CS-86-134 (PhD Thesis), Computer Science Department, CMU (May 10, 1986).
21. C. A. R. Hoare, ''Towards a Theory of Parallel Programming'', pp. 61-71 in *Operating system techniques*, Academic Press (1972).
22. P. Brinch-Hansen, *Operating System Principles*, Prentice-Hall, New Jersey (July 1973).
23. Ada Run Time Environment Working Group, ''A Catalog of Interface Features and Options for the Ada Run Time Environment Release 3.0'', *Ada Letters* (to appear 1991).
24. A. J. Wellings, *Measuring, Monitoring and Enforcing CPU Execution Time Usage - ARTEWG Study Note*, Department of Computer Science, University of York (August 1991).