Tool Support for the Construction of Statically Analysable Hard Real-Time Ada Systems*

T. Vardanega
Spacecraft Control and Data Systems Division
European Space Agency Research & Technology Centre (ESTEC)
Keplerlaan 1, 2201 AZ Noordwijk (NL)
voice: +31-71-5655331 fax: +31-71-5654295
email: tullio@wd.estec.esa.nl

Technical Note

Abstract

This paper maintains that fixed-priority process-based preemptive scheduling is, arguably, more convenient, flexible and responsive than conventional cyclic scheduling for the construction of new-generation software-intensive satellite control systems. Predictable usage of preemptive priority-based scheduling, however, demands the support of mature static analysis techniques. Worst-case response time analysis models can be constructed which minimise the embodied pessimism and maximise useful processing. This paper presents the design and implementation of an Ada programming model and associated worst-case response time analysis tools aimed to support the construction of highly-predictable, highly-efficient on-board control systems.

1 Introduction

Recent studies (cf eg: [12]) have shown that software embedded on board of modern satellite control systems plays an increasingly important and pervasive role in the operation of the system. Distinct demands for increased responsiveness and maximised mission product, in fact, call for the progressive move of critical functions from ground to on-board software. As a result of this evolution, new-generation on-board systems appear to be:

- increasingly *concurrent*, as they are to perform, in parallel, a growing variety of control activities featuring a broad range of activation and processing requirements,
- and distinctly *hard real-time*, in that an important proportion of their software components are subject to mission-critical requirements on timeliness of execution.

This paper presents the principal technical choices made by a project aimed at supporting the development of hard real-time multi-tasking on-board software systems. The project considered traditional development and scheduling methods as being too poor, rigid and unyielding in the face of the flexibility and

*The work described in this paper has been performed under ESA Research & Technology Center contract no. 9848/92/NL/FM. An executive summary of this paper is to appear in the proceedings of the 17th Real-Time Systems Symposium (RTSS'96).

responsiveness required of future systems (cf eg: [8]) and also largely unsupportive of sufficient functional cohesion. Novel engineering techniques for the construction, analysis and execution of hard real-time systems, on the other hand, had consistently emerged from the research community over the last decade (cf eg: [3, 4, 5, 10] and seemed mature enough for deployment.

The selected approach, which retained Ada [2] as the programming language of choice, was centred around fixed-priority preemptive scheduling augmented with priority ceiling emulation [5] and worst-case response time analysis [1, 7, 10]. (The rationale for this choice is discussed at length in [11].) The feasibility and performance of the chosen approach were initially demonstrated by a couple of ESA-funded case studies (cf [13, 14, 19]) and a few other independent industrial developments.

Use of Ada tasking in space and avionic applications was historically tied to the criticisms of being too complex and non-deterministic and relying on too extensive and inefficient run-time support. The project's initial results seem to dismiss a great deal of such criticisms. This paper presents the distinguishing features of the proposed concept and its anticipated benefits.

The rest of the paper is structured as follows: section 2 describes components, properties and implementation of the chosen programming model and the associated tool-set concept; section 3 discusses the requirements on and the definition of the tools and techniques for the support of static scheduling analysis; section 4, finally, presents the current status with the tool-set implementation and outlines its projected scope of application.

2 Programming Model

2.1 Programming Model Definition

Typical on-board applications (cf eg: [11]) are predominantly comprised of non-independent activities which exhibit a varying amount of periodic and sporadic activation requirements and an equally varied degree of timing criticality.

Attitude control functions map to periodic activities whose activation needs to be as jitter-less as possible. Communication control functions map to sporadic activities whose activation requirements stem from both external (eg: interrupts) and internal events (eg: synchronisation). Periodic and sporadic activities often require some form of explicit cooperation to achieve data-oriented synchronisation and/or enforcement of precedence activation constraints.

Past and current evidence show that such an application model, albeit with a certain amount of bending and twisting, may indeed be implemented upon a single-process cyclic scheduling system. It is equally apparent, though, that the same application model would also nicely fit a simple and yet slightly augmented form of Ada tasking based on the Ada 83 concurrency model, now fully supported by the revised Ada standard [18].

An Ada programming environment specialising in the support of such an application model has been built for on-board systems based on the 32-bit Embedded Real-Time Computing Core (ERC32) [16]. The ERC32 core is a SPARC v7 based chipset inclusive of Integer Unit, Floating Point Unit and Memory Controller, intended for use in *no-cache no-MMU* single-board computers for highly predictable high-profile new-generation on-board systems. Amongst other features, which are not discussed in this paper, the ERC32 chipset and associated programming environment are designed to allow exploitation of the ATAC (Ada Tasking Coprocessor) chip [9], a memory-mapped hardware device which performs Ada 83 tasking operations on behalf of classical software run-time systems.

In the ERC32 programming model, every distinct processing activity in the application model is modelled as an independent thread of control and maps to one designated type of Ada task. Tasks in the model are required to be library-level, flat, static, infinite tasks. Three types of task are required to support the application model:

- *cyclic* tasks to model periodic activities;
- *interrupt sporadic* tasks to model interrupt-driven sporadic activities;
- *software sporadic* tasks to model synchronisation-driven sporadic activities.

The typical profiles for each of such task types are as shown in template 2.1.1, 2.1.2 and 2.1.3 in the following. The statements in bold-face in the profiles highlight the places at which the task's execution requires explicit run-time system support. This information will be used later in this paper to illustrate the foundations to and the operation of the chosen approach to static scheduling analysis.

Template 2.1.1 *Periodic Task Template*

```
task CYCLIC is
  pragma priority (< value >);
end CYCLIC;

task body CYCLIC is
  – declare local variables
begin
  – set up application-wide start time To
    loop
      Delay_Until T ;
      – execute periodic actions
      T := T + PERIOD;
    end loop;
end CYCLIC;
```

The interrupt model supported by the programming model was designed to take maximum advantage of the (optional) presence of the ATAC chip. In the adopted model, a library-level parameterless procedure is to be attached to the designated interrupt line. In presence of the ATAC, the procedure would be compiled in an ATAC-internal dummy task. The ATAC would then capture the incoming interrupt, activate the dummy task, ready the interrupt sporadic task and initiate the rendez-vous between them.

In absence of the ATAC, the interrupt would be captured by the ERC32 Memory Controller and cause the software run-time to initiate the accept statement in the interrupt sporadic task.

In order for both options to be completely interchangeable without the need for any source code modification, no user code is allowed in the library procedure.

Tasks in the model are not allowed to communicate directly with one another. Communication and synchronisation between tasks take place via dedicated server tasks which can be implemented as Ada 95 protected objects as well as by Ada 83 passive tasks. This choice buys the system an important reduction in the required run-time support and, thanks to the use of *immediate priority ceiling inheritance* (IPCI) [5], it also defeats non-determinism by guaranteeing bounded blocking and priority-inversion-free service.

- Tasks exchange shared data in a protected manner by means of mutually-exclusive calls to dedicated *resource* server tasks; resource server tasks may offer a variety of services, each represented by one distinct entry; entries to a resource server task must rigorously be unguarded (cf profile 2.1.5).

Template 2.1.2 *Interrupt Sporadic Task Template*

```
task INTERRUPT_SPORADIC is
   pragma priority (An_Interrupt_Priority);
      – System.Interrupt_Priority range 49 .. 63
   entry ISR;
      for ISR'Address use TRAP_NUMBER;
         – Interrupt_Manager.Interrupt_Id range 1 .. 15
end INTERRUPT_SPORADIC;

task body INTERRUPT_SPORADIC is
   – declare local variables
begin
   loop
      accept ISR do
         – immediate interrupt service
         – executed at priority Interrupt_Manager.Interrupt_Id
      end ISR;
      – interrupt service processing
      – executed at priority An_Interrupt_Priority
   end loop;
end INTERRUPT_SPORADIC;

procedure LIBRARY_LEVEL is
– must be parameterless to ensure ATAC compatibility
begin
   INTERRUPT_SPORADIC.ISR;
end;

Interrupt_Manager.Attach_Handler
   (HANDLER => LIBRARY_LEVEL'Address,
   INTERRUPT => TRAP_NUMBER,
   PRIORITY => An_Interrupt_Priority,
   ... );
– procedure LIBRARY_LEVEL must be explicitly attached to designated trap
```

- Tasks synchronise with one another by means of mutually-exclusive calls to dedicated *synchronisation* server tasks; the model requires individual synchronisation server tasks to provide one guarded entry for exclusive use by the designated software sporadic task and one unguarded entry for use by the releasing task(s) (cf profile 2.1.4).

Template 2.1.3 *Software Sporadic Task Template*

```
task SOFTWARE_SPORADIC is
   pragma priority (< value >);
end SOFTWARE_SPORADIC;

task body SOFTWARE_SPORADIC is
   – declare local variables
begin
   – set up application-wide start time To
      loop
         SYNCHRONISATION_PO.WAIT (< parameters >);
         – execute sporadic action
      end loop;
end SOFTWARE_SPORADIC;
```

2.2 Programming Model Implementation

Implementation of the ERC32 programming model as above outlined requires a number of distinct enhancements to a conventional Ada 83 compilation system (ACS). The principal modifications and distinguishing features of this implementation are briefly discussed in the following:

Support for ERC32 Targets with and without ATAC :
ERC32 target boards may include the Ada Tasking Coprocessor (ATAC) chip; the ERC32 ACS supports the presence of the ATAC and is able to devolve the tasking operations to the ATAC; the ERC32 programming model is designed so as to achieve source-code invariance across targets with or without the ATAC: the final decision as to whether to run on the ATAC run-time or stay with the software run-time is made at bind time.

Support for Deadline-Monotonic Scheduling :
The ERC32 ACS's main aim is to support the construction of statically analysable systems built in accordance with deadline-monotonic scheduling (DMS) theory; the provided support includes:

Passive Task Optimisation :
The ERC32 ACS provides support for passive task optimisation; this is achieved by inclusion of **pragma passive** in the relevant task's specification which, in turn, results in compile-time checks on the passive task structure (including enforcement of related usage restrictions) and the generation of non-tasking light-weight run-time system implementation of its dynamic semantics; priority-inversion-free mutually exclusive access to passive tasks is achieved by use of IPCI.

Worst-Case Execution Profile Extraction and Timing Data File :
The ERC32 ACS supports automated extraction of worst-case execution time (WCET) of (por-

Template 2.1.4 *Synchronisation Passive Task Template*

```
task SYNCHRONISATION_PO is
   pragma priority (Ceiling_Priority);
   pragma passive;
   entry WAIT (< parameters >);
   entry SIGNAL (< parameters >);
end SYNCHRONISATION_PO;

task body SYNCHRONISATION_PO is
   -- declare local variables
begin
      loop
         select
            when (Barrier) =>
               accept WAIT (< parameters >) do
                  -- release actions
         or
            accept SIGNAL (< parameters >) do
               -- lower Barrier
         or
            terminate;
         end select;
      end loop;
end SYNCHRONISATION_PO;
```

Template 2.1.5 *Resource Control Passive Task Template*

```
task RESOURCE_PO is
   pragma priority (Ceiling_Priority);
   pragma passive;
   entry SERVICE (< parameters >);
   ...
end RESOURCE_PO;

task body RESOURCE_PO is
   -- declare local variables
begin
      loop
         select
            accept SERVICE (< parameters >) do
               -- service actions
         or
            ...
         or
            terminate;
         end select;
      end loop;
end RESOURCE_PO;
```

tions of) Ada programs; this includes: 1/ provision of options to restrict WCET-defeating code-motion optimisations; 2/ enforcement of programming-model coding restrictions and associated templates; 3/ automated generation, storage and retrieval of the timing information describing the worst-case execution profile of the application (aspects of generation and analysis of such profiles are discussed in section 3.3); 4/ provision of timing measurements for the set of run-time system primitive needed to support the tasking operations allowed by the programming model: the set of required primitives, derived from analysis of the task templates in section 2.1, is reported in table 1.

Time-Bounded Run-Time Subset and Interval Timer Model :
The set of ERC32 ACS run-time system primitives and tasking operations and the associated data structures have been re-designed so as to guarantee *minimum deterministic* worst-case execution time.
The ERC32 ACS run-time system supports the *Interval Timer* model instead of the conventional Periodic Clock model. In the Interval Timer model, interrupts off the real-time clock are triggered only as suspended tasks need to be readied. This buys the application a significant reduction of run-time system overhead.

Schedule Trace :
The ERC32 ACS run-time supports optional selective tracing of task scheduling activity with the trace data stored in an in-memory circular buffer; enable/disable of tracing is performed both from the debugger and via an ad-hoc programmatic interface; the size of the trace buffer is set via a binder option.

Package Real_Time :
The ERC32 ACS provides a user-visible run-time support package containing **type Time**, supporting the 1 μs-accuracy ERC32 real-time clock, the associated monotonic time representation with related relational and arithmetics operators and general-purpose timers. Periodic activities which require jitter-less release must call the **Delay_Until**(<absolute time>) procedure provided by the package in preference to the standard Ada 83 **delay** (<relative time>), as shown in task template 2.1.1.

The ERC32 programming model was implemented on top of a small, highly efficient and fully characterised Ada run-time system. All of the required tasking primitives were designed so as to provide for optimised bounds to worst-case execution time,

In the following, a brief description is provided for the tasking primitives which contribute to the determination of the run-time scheduling behaviour of the ERC32 system (cf table 1).

Cyclic tasks call primitive Delay_Until to command the time of their next release and the wake-up system uses an Interval Timer instead of the conventional periodic clock. The overall worst-case execution time of the primitive results from the sum of two values: the placement of the task control structure in the interval time queue (**Delay_Until(Enter)**) and the return from the call upon release (**Delay_Until(Exit)**).

Interrupts off the Interval_Timer are serviced by primitive **Timer_Interrupt**. Primitive **Ready_After_Delay** change the released cyclic tasks' status to ready. On modifications to the ready status list, primitive **Sched_Select** is invoked to determine the "best-task-out"; this may incur preemptive switch to a new running task, which is performed by primitive **Context_Switch**.

Primitive **Interrupt_Handling** initiates an interrupt accept statement in the body of the designated interrupt sporadic task, while **Interrupt_Sporadic_Wait(Enter)** and **Interrupt_Sporadic_Wait(Exit)** allow control to respectively enter and leave the interrupt accept body.

Table 1: *Required Run-Time System Primitives*

Primitive RTS Operation	Required for Task Template
Delay_Until (Enter at Top)	2.1.1
Delay_Until (Enter Lower)	2.1.1
Delay_Until (Exit)	2.1.1
Timer_Interrupt	2.1.1
Ready_After_Delay	2.1.1
Sched_Select	2.1.1, 2.1.2, 2.1.3
Context_Switch	2.1.1, 2.1.2, 2.1.3
Interrupt_Handling Overhead	2.1.2
Int_Sporadic_Wait (Enter)	2.1.2
Int_Sporadic_Wait (Exit)	2.1.2
PO_Entry	2.1.3, 2.1.4, 2.1.5 (, 2.1.1, 2.1.2)
PO_Exit	2.1.3, 2.1.4, 2.1.5 (, 2.1.1, 2.1.2)
Sem.Wait (Enter)	2.1.3, 2.1.4
Sem.Wait (Exit)	2.1.3, 2.1.4
Sem.Signal	2.1.4
Entry_Queue_Mgmt	2.1.3, 2.1.4

PO_Entry and **PO_Exit** control respectively the access to and the release of server tasks and include the relevant raising and lowering of the server's priority.

The blocking call to a synchronisation server task is implemented by use of a primitive semaphore structure: the software sporadic task's call to one server's guarded entry translates into the caller's suspension on the primitive semaphore (**Sem.Wait(Enter)**). Arrival of the releasing call causes the suspended task to be freed from the semaphore's queue (**Entry_Queue_Mgmt**, which *includes* call to **Sem.Signal**), exit from the suspensive call (**Sem.Wait(Exit)**) and potentially become the new running task.

Primitive Sched_Select involves queue management operations which are prone to pessimistic bounds; the primitive was, therefore, redesigned so as to preserve minimal execution time and also achieve low worst-case bounds. The design restriction of having at most one software sporadic task wait on any given synchronisation server's semaphore queue allows all of the relevant primitive operations to be easily bounded. All other primitives in the above list naturally feature completely deterministic execution time bounds. The characterisation of all such bounds on execution on the selected platform is stored in the so-called *run-time system characterisation file* (RCF).

2.3 Toolset Concept

The ERC32 programming model as above presented is supported by an integrated set of static analysis tools. The block-level description of the tool-set concept is shown in figure 1.

Ada programs which comply with the ERC32 programming model requirements are compiled by the ERC32 Ada compilation system (ACS). Worst-case execution profiles are automatically generated, for all tasks in the program, by a specially-designed enhancement to the the ACS, the ESF generator, whose implementation was eased by by the distinguishing features of the ERC32 programming model (eg: no Ada constructs allowed which have no time-boundable operation, such as heap management and dynamic task creation) and execution platform (eg: no cache and no MMU allowed).

In the generation of such profiles (collectively termed the program's *Execution Skeleton*, ESF), the ESF

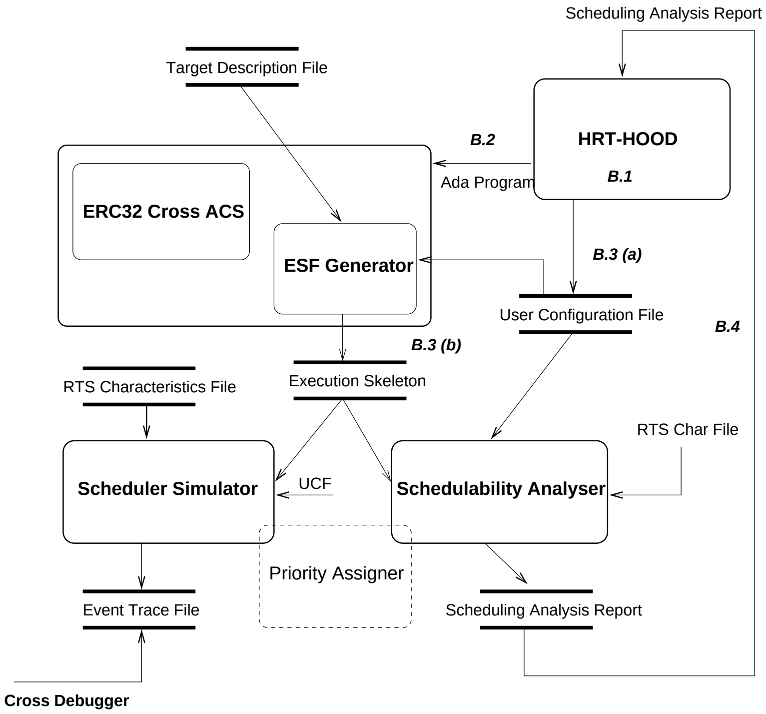

Figure 1: *Toolset Concept Logical Model*

generator uses information from the following sources: (i) compiler's internal data structures for path analysis and worst-case selection; (ii) configurable look-up tables describing the execution cost of the ERC32 instruction set (including guidance to resolve data-dependent computation time estimates) under the chosen board configuration; (iii) a user-provided description of the static hard real-time attributes of all cyclic, interrupt sporadic and software sporadic tasks in the system, *User Configuration File* (UCF); the relevant attributes include *type* (ie: cyclic, interrupt sporadic, software sporadic), *criticality* (ie: interrupt, hard, soft, non-critical), *period* (or minimum interarrival time) and *deadline*. Issues with the generation of the ESF are discussed in section 3.3.

Priority assignment is performed on static analysis of the UCF and ESF. Tasks are *not* allowed to share the same priority level. Tasks with decreasing criticality are assigned decreasing priority levels, whereas tasks within the same criticality range are assigned priority levels in deadline-monotonic fashion.

Server tasks do not possess user-assigned static attributes as they just inherit a ceiling priority from their callers and, therefore, need not appear in the UCF. Server tasks' ceiling priority is set at least one level higher than the maximum priority of callers.

The program's ESF, together with the UCF and the RCF, is processed by the Scheduling Analyser tool to determine the tasks' worst-case response time. Elements of the undertaken analysis are discussed in section 3.1.

The use relationship between caller tasks and associated servers is captured by the compiler in the generation of the Execution Skeleton. Furthermore, all types of caller-server use relationships in the system as well as the complete set of user-assigned hard real-time attributes can also be automatically extracted from software design tools which support the HRT-HOOD design method [15]. The HRT-HOOD method was, in fact, defined in the frame of preparatory activities to the ERC32 project and is fully consistent with the ERC32 programming model. The implementation of a prototype HRT-HOOD design support tool capable of automated generation of UCF and equally automated extraction of tasking structures conforming with the designated profiles has recently been completed (cf [20]) and is presently undergoing evaluation.

The program's Execution Skeleton may also be processed by the Scheduler Simulator tool. This tool performs event-driven simulation of the run-time scheduling behaviour of the system and returns selective trace of the scheduling events occurred over a given time duration. The event trace produced in this manner may then be compared, for test and verification purposes, with the trace optionally generated during real runs of the system.

Profile 2.3.1 shows the UCF and ESF syntax required to describe, for exampe, a cyclic task conforming with template 2.1.1.

Profile 2.3.1 *Model of CYCLIC for Static Analysis*

```
---------------
----------- UCF
---------------
THREAD Template.Cyclic
 CRITICALITY hard | soft | not_critical
 PERIOD   <time>
 DEADLINE <time>
 OFFSET   <time>
END Template.Cyclic
---------------
----------- ESF
---------------
THREAD Template.Cyclic
 TYPE CYCLIC
 PRIORITY <integer>
 -- processing profile
   -- periodic actions
   { [ WCET <time> | CALL_PO <PO name> <entry name> ] }
 -- end processing profile
 [comprehensive list of called POs (entry level)]
END Template.Cyclic
```

3 Enabling Static Analysis

3.1 Foundations of Response Time Analysis

The static analysis model chosen for the ERC32 system concept aims at the prediction of worst-case response times (cf eg: [1, 7]). The model stipulates that one thread's worst-case response time be defined as the longest elapsed time it takes for that thread to complete its most demanding set of activities in response to an activation occurring under maximum contention from the rest of the system. (The term thread is used in the following as a synonym for task.) The worst-case response time of any thread τ_i does, thus, result from suitable combination of the following three distinct components:

(i) The *worst-case computation time* of thread τ_i, $\mathbf{WCCT}_i$, which is defined as the sum of the time cost of all τ_i's sequential blocks of execution which lay in the statically determined worst-case path enclosed within the thread's main loop (the thread's execution profile), in addition to the time cost of the run-time system services required for the support of that execution.

(ii) The *interference* incurred by τ_i, $\mathbf{I}_i$, which is caused by the occurrence of preemptive execution of higher-priority threads and higher-priority run-time system services incurred during τ_i's ready period; in the ERC32 model, the interference from the run-time system is limited to the handling of the interrupts off the Interval Timer, as all other interrupts are tied to the run of interrupt sporadic tasks.

(iii) The *blocking* experienced by τ_i, $\mathbf{B}_i$, which originates from the possibility that a due release of τ_i be delayed by other effects than those arising from preemptive interference; such effects occur when the run-time system protects the execution of internal critical sections by temporarily inhibiting (ie: deferring) preemption as well as a consequence of adopting IPCI for the implementation of mutual exclusion in the communications between tasks and servers; use of IPCI may, in fact, delay the release of tasks whose priority is higher than the caller but lower than the server's ceiling; response time analysis prescribes that worst-case blocking be determined as the *largest* possible delay effect incurred from any of the two sources.

For any thread τ_i, component $\mathbf{WCCT}_i$ is fully determined at compile time on the Ada closure of the program, component $\mathbf{B}_i$ is a function of the assigned priorities and the system's run-time performance, and component $\mathbf{I}_i$ is a function of the system load.

Component $\mathbf{WCCT}_i$ and $\mathbf{B}_i$ are maximised by analysis. Care must be taken, though, to avoid incurring excessive pessimism in their determination, as this may hinder the usefulness of the analysis. Section 3.2 presents the approach taken to the determination of $\mathbf{B}_i$, whilst section 3.3 discusses issues in the generation of the worst-case execution profiles from which $\mathbf{WCCT}_i$ is determined.

Component $\mathbf{I}_i$ is maximised by assuming all runs to occur under the notional concept of *critical instant*:

- all cyclic tasks are assumed to be disjointly released at time $t_0 = 0$
- all interrupt sporadic tasks are assumed to be disjointly triggered at time $t_0 = 0$ and arrive at their maximum frequency
- all software sporadic tasks are assumed to be released off their synchronisation server's queue at time $t_0 = 0$.

The formulae which capture the interference effects on τ_i's ready period over the interval $[0, t)$ are shown in the following, where notation $j \epsilon HP(i)$ denotes that thread τ_j;s priority is greater than τ_i's (ie: $Pr(\tau_j) \geq Pr(\tau_i)$), notation $j \epsilon LP(i)$ denotes the converse (ie: $Pr(\tau_j) < Pr(\tau_i)$), and T_j denotes τ_j's period (for cyclic threads) or minimum interarrival time (for sporadic threads):

Interference Effect due to Preemptive Execution of Higher-Priority Threads over [0,t) :

$$\mathbf{I}_i^t = \sum_{j \epsilon HP(i)} \lceil \frac{t}{T_j} \rceil \mathbf{WCCT}_j \tag{1}$$

Interference Effect due to Interrupts Off the Interval Timer over [0,t) :

$$\mathbf{I}_{clk}(i, t) = K(i, t) * (Timer\_Interrupt + Ready\_After\_Delay + Sched\_Select) \tag{2}$$

In ATAC mode (equation 3), the ATAC filters out all Timer interrupts and *only* releases those which induce preemption:

$$\mathbf{K}_i^t = \sum_{j \epsilon HP_{CYCLIC}(i)} \lceil \frac{t}{T_j} \rceil \tag{3}$$

In non-ATAC mode (equation 4), there occurs no filtering and the calculation needs to consider also the *first*, non-preemptive, critical-instant release of lower-priority cyclic threads:

$$\mathbf{K}_i^t = \sum_{j \epsilon HP_{CYCLIC}(i)} \lceil \frac{t}{T_j} \rceil + \sum_{k \epsilon LP_{CYCLIC}(i)} 1 \tag{4}$$

The ERC32 tool-set supports two variants of analysis techniques. The first variant, based on deadline monotonic theory [7, 10] (DMS), assumes that tasks' deadlines cannot exceed the respective period (or minimum interarrival time) and determines, for every individual thread, the response time for a *single* critical-instant release. The other variant, based on the extended version of deadline monotonic theory presented in [17] (ADS), assumes that deadlines may be arbitrarily greater than the relevant period (or minimum interarrival time) and, therefore, extends the solution space to *multiple, overlapping* releases of a task. The critical-instant assumptions are, thus, worsened by tasks' releases being delayed past their due time also by the outstanding completion of their previous releases.

The equations for DMS and ADS response time analysis are shown in the following. Both are based on recurrence relations in which thread τ_i's response time, R_i^n, is expressed as a monothonically increasing summation term. The DMS recurrence is somewhat simpler than its ADS variant and guaranteed to converge when the system's utilisation is not greater than 1. The ADS variant, albeit based on the same conceptual model as DMS, is slightly more complex as its solution space extends across multiple, overlapping releaes; as shown by equation 12, the search stops as soon as the response time for the last release no longer overlaps the next due activation.

Response Time (DMS) :

$$\mathrm{R}_i^n = \mathbf{B}_i + \mathbf{WCCT}_i + \mathbf{I}_i^{\mathrm{R}_i^{n-1}} + \mathbf{I}_{clk}(i, \mathrm{R}_i^{n-1}) \qquad (n > 1) \tag{5}$$

$$\mathrm{R}_i^1 = \mathbf{B}_i + \mathbf{WCCT}_i \tag{6}$$

Response Time (ADS) :

Busy Window at (q+1)th release:

$$\mathrm{R}_i^{n+1}(q) = \mathbf{B}_i + (q+1)\mathbf{WCCT}_i + \mathbf{I}_i^{\mathrm{R}_i^n(q)} + \mathbf{I}_{clk}(i, \mathrm{R}_i^n(q)) \qquad (q, n \geq 0) \tag{7}$$

where:

$$\mathrm{R}_i^0(q) = \mathrm{R}_i(q-1) \qquad (q \geq 1) \tag{8}$$

and

$$\mathrm{R}_i^0(0) = \mathbf{B}_i + \mathbf{WCCT}_i \tag{9}$$

Response Time at (q+1)th release:

$$\mathrm{R}_i(q) = \mathrm{R}_i^n(q) - qT_i \tag{10}$$

Worst-Case Response Time:

$$\mathrm{R}_i = \max_{q \epsilon N} \mathrm{R}_i(q) \tag{11}$$

where:

$$N = setof(q) : \mathrm{R}_i^n(q) > (q+1)T_i \Longrightarrow \mathrm{R}_i(q) > T_i \tag{12}$$

3.2 Blocking Overhead Determination

The worst-case blocking effect incurred on one thread's release is determined as the largest value between the single longest period of run-time deferred preemption and the longest-duration entry call to a higher-ceiling server performed by a lower-priority task.

The former value is a *constant* characteristic of the run-time system implementation. In the case of the ERC32 ACS, this value is minimised by the restrictions imposed on the ERC32 programming model.

The latter value is a *variable* thread-specific attribute which depends upon such application-wide characteristics as the assigned priorities and the performance of servers' entries. The pessimism potentially embodied in the determination of this value is minimised by the analysis which follows.

In the ERC32 programming model, calls to server tasks' entries must conform to any of the types shown in table 2.

Table 2: *Types of PO Calls*

server type	entry type	call denotation
resource server (RPO)	unguarded service call	RPO.Service
synchronisation server (SPO)	unguarded releasing call	SPO.Signal
synchronisation server (SPO)	guarded suspensive call	SPO.Wait

The contribution to one thread's blocking overhead resulting from use of IPCI is determined by the largest execution cost of any of the above entry calls. This includes the $\mathbf{WCCT}$ of the selected accept body and the execution cost of all the run-time system operations needed to service the selected entry call.

Calls to resource servers are unconditional, hence incur a constant run-time system overhead. Calls to synchronisation servers are guarded, hence exhibit worst and best execution profiles: a guarded suspensive call may find the guard *open* (best case) and incur no suspension, or *closed* (worst case) and incur suspension and deschedule; an unguarded releasing call (SPO.Signal) may find *no* awaiting task in the entry queue (best case) and let the caller continue, or *one* awaiting task (worst case) and cause its release off the queue and, possibly, the deschedule of the caller. Table 3 lists the *individual* overhead components which are incurred while executing the PO calls shown in table 2 under both worst and best cases.

Table 3: *PO Call Overhead*

call type	execution components	ID	case type
RPO.Service	PO_Entry + **WCCT**(Service) + PO_Exit	(1)	worst,best
SPO.Signal	PO_Entry + **WCCT**(Signal) + + Entry_Queue_Mgmt + Context_Switch	(2)	worst
SPO.Wait (enqueue)	PO_Entry + (guard_eval) + Sem.Wait (Enter) + + Sched_Select + Context_Switch	(3.1)	worst
SPO.Wait (dequeue)	Sem.Wait (Exit) + (guard_eval) + + **WCCT**(Wait) + PO_Exit	(3.2)	worst
SPO.Signal (no_wait)	PO_Entry + **WCCT**(Signal) + PO_Exit	(4)	best
SPO.Wait (open)	PO_Entry + (guard_eval) + **WCCT**(Wait) + + PO_Exit	(5)	best

It should be noticed that the two worst-case components of an SPO.Wait call occur at *two separate points in time*: component (3.1) occurs from the caller's entering of the SPO to caller's suspension and placement in the entry queue; component (3.2) occurs from caller's release from the entry queue to caller's departure from SPO and *always* directly follows execution of component (2).

Response time analysis based on DMS considers one single critical-instant release of the thread subject of analysis and also requires that all guards encountered in that run be initially closed. DMS analysis is, thus, interested in worst-case service values only. Analysis based on ADS contemplates multiple, potentially overlapping releases and may, therefore, need to consider best-case values, too. For example, a software sporadic thread τ_i performing a guarded suspensive call at its q-th release (with $q > 0$) will find the guard open if τ_i's priority is *lower* than the releasing thread's. Similarly, thread τ_i performing an unguarded releasing call will find no awaiting thread in the entry queue if τ_i's priority is *higher* than that of the software sporadic thread associated with that synchronisation server.

Column 5 of Table 4 prescribes how the individual overhead components listed in Table 3 contribute to the bound for the IPCI blocking on thread τ_i: tag L under column 2-4 denotes that $\exists k : k \epsilon S(j) : j \epsilon LP(i) \wedge$ *calls*$(j, PO.Call) \wedge PO \epsilon HP(i) \wedge \mathbf{WCCT}_k(Call) = max(\mathbf{WCCT}_{j \epsilon S}(Call))$ computed over all the entry calls of every individual server in the application; tag H denotes that the thread set $S(j)$ is empty for that particular type of server call. The definitive bound on the IPCI blocking is then calculated as the maximum value amongst those captured by Table 4.

Table 4: *Call Overheads Accountable for Blocking*

	call type			blocking factor	
	RPO.Service	**SPO.Signal**	**SPO.Wait**	**Worst Case**	**Best Case**
Pr(Caller) vs Pr(τ_i)	H	H	H	none	none
	L	H	H	(1)	(1)
	L	L	H	max(1,2)	max(1,4)
	L	L	L	max(1,(2+3.2),(3.1))	max(1,4,5)
	H	L	L	max((2+3.2),(3.1))	max(4,5)
	H	H	L	(3.1)+(3.2)	(5)
	H	L	H	(2)	(4)
	L	H	L	max(1,((3.1)+(3.2))	max(1,5)

3.3 Worst-Case Execution Profile Generation

The generation of the ESF from the application's source code must attempt to capture both the *local* worst-case at thread-level and the *global* worst-case at application-level in a manner which incurs a controlled degree of induced pessimism.

Excessive pessimism may arise, for example, when the resolution of a branch or the bounding of an iteration within one thread's profile fail to capture application-wide path exclusion conditions (eg: mutually exclusive operating modes) or run-time best-bounding information. This may cause otherwise provably impossible paths to be selected and consequently yield too conservative predictions.

The ESF generator attempts to mitigate such problems by providing means for the user to annotate the source code with a *loop-bound* and a *path-exclusion* pragma: (i) **pragma Loop_Count** ($< constant >$) placed before a for or while loop construct allows the user to supply the preferred bound to an otherwise unbound iteration; the compiler uses the provided bound value to cost the iteration but returns warnings if it was able to statically determine a better bound; (ii) **pragma Exclude_Wcet** placed inside a conditional branch, procedure body or task body causes the exclusion of the tagged construct from the selected path.

The primary goal of the ESF generator is to determine, for every thread τ_i in the system, the execution profile which maximises the $\mathbf{WCCT}_i$ component amongst the alternate control flow paths allowed by the User's placement of the above pragmas.

Thread τ_i's $\mathbf{WCCT}_i$ is made up of two components: (i) τ_i's own execution cost as determined at compile time by the ESF generator and (ii) the run-time system overhead incurred by τ_i in the execution of *one* release in the selected path. The latter component depends on the thread's type and the server calls retained in the profile. The relevant overheads are listed in table 6.

The achievement of justified maximisation of $\mathbf{WCCT}_i$, however, is not the sole objective of the ESF generator. There, in fact, exist two distinct ways for thread τ_i to effect system's responsiveness:

- a longer $\mathbf{WCCT_i}$ induces a longer $\mathbf{I}_j$ on thread $\tau_{\mathbf{j}} \forall j : j \epsilon LP(i)$;
- a PO.Call performed by τ_i may contribute to $\mathbf{B}_k$ for thread τ_k $\forall k : k \epsilon HP(i) \wedge k \epsilon LP(PO)$.

The ESF generation algorithm must, therefore, also seek to achieve justified maximisation of $\mathbf{B}_k$.

Consider the code fragment in example 3.3.1 and assume that $< condition >$ cannot be statically resolved. The example shows a classical case in which *local* maximisation of $\mathbf{WCCT}_i$ may degrade the determination of $\mathbf{B}_k$ for any thread τ_k in the system. This case occurs on the selection of branch B, when no trace of PO.Call is retained in the thread's profile.

The problem is resolved, in the ERC32 system, by instructing the branch selection algorithm to keep record of all the server calls performed *outside* the retained profile and to require that *alternate* blocking analysis be performed of the potential blocking effect of such calls. This analysis may possibly yield a larger $\mathbf{B}_j$ value for some thread τ_j, thereby highlighting a potential conflict between *local* and *global* worst-case path selection criteria. In any such case, the server call responsible for thread τ_j's alternate blocking is identified to the User and the User is advised to consider repeating the analysis forcing the extraction of execution profiles which include that server call.

```
if <condition> then
   -- branch A
   <A1> -- sequential block
   PO.Call
   <A2> -- sequential block
else
   -- branch B
   <B>  -- sequential block
end if;
```

Example 3.3.1: *Effect of Blocking on Path Selection*

Sequential blocks of execution are delimited by explicit server calls, as their processing require execution of run-time system code, and start of bounded iterations, as these cannot be unfolded for the sake of unifying the input ESF for both Scheduler Simulator and Scheduling Analyser.

All the $< time >$ values specified in WCET-type profile statements are determined by the ESF extractor by summation of the execution cost of the assembly instructions enclosed within the boundaries of the relevant source block and exclusively belonging to the thread's own code. The execution cost of the individual assembly instructions is specified in the Target Characteristics File for the chosen ERC32 board configuration (cf figure 1). One thread's $\mathbf{WCCT}$ is, thus, computed by summation of all the WCET $< time >$ components included in the thread's profile *plus* the cost of all the run-time system services required for management and administration support of the thread's operation.

3.4 Preliminary Characterisation of Run-Time System Overheads

Preliminary characterisation of the worst-case run-time system overheads was performed upon completion of the the ERC32 ACS implementation. Whereas the ultimate objective in this respect is to perform the characterisation using the timing capabilities embodied by the ESF extractor, the *initial* measurements were taken out of real runs on an ERC32 preliminary demonstration board, artificially enforcing worst-case execution paths wherever necessary.

The performed measurements also included determination of the longest deferred-preemption time incurred during run-time system operation, as required for analysis of blocking overheads.

The measurements were performed on ERC32 configurations with *and* without the ATAC, but the results are not directly comparable with one another as the former reflect a presently complete and highly optimised implementation while the latter are only qualitative and still awaiting completion of testing and optimisation. The results from this preliminary characterisation are shown in table 5

Table 5: *Timing Characteristics of Basic Run-Time System Operations (μs)*

RTS Operation	**DEM32 (10 MHz 0 Wait-states)**	
	non-ATAC run-time	**ATAC run-time**
PO_Entry	8.0	14.0
PO_Exit	11.0	11.0
Int_Sporadic_Wait (Enter)	3.0	3.0
Int_Sporadic_Wait (Exit)	3.0	0.0
Sem.Wait (Enter)	7.0	7.0
Sem.Wait (Exit)	3.0	3.0
Sched_Select	5.0	0.0
Context_Switch	34.0	41.0
Delay_Until (Insert at Top)	39.0	23.0
Delay_Until (Insert Lower)	$22.0 + \mathbf{C} * 3.0$	23.0
Delay_Until (Exit)	8.0	8.0
Entry_Queue_Mgmt	6.0	8.0
Ready_After_Delay	12.0	0.0
Timer_Interrupt	21.0	0.0
Interrupt_Handling_Overhead	67.0	0.0
Max_Deferred_Preemption	130.0	65.0

In accordance with the earlier discussion in this paper, the *only* parametric expression in table 5 is the one which describes the cost of placing a cyclic thread in the delay queue at a position lower than the top. The actual position depends on the relative ordering of the required awake time by the Interval Timer. Term **C**, thus, denotes the total number of cyclic threads currently placed ahead of the presently suspending thread. The best value for **C**, for use by static analysis, obviously depends on the knowledge available to the tool in question. In the case of the ERC32 tool-set, this value shall be the *exact* number of presently ahead-enqueued threads, for the Scheduler Simulator, and the *total* number of cyclic threads in the system, for the Schedulability Analyser.

The information contained in table 5, in conjunction with the analysis of the ERC32 ACS implementation discussed earlier in this paper, allows the run-time system contribution to one thread's $\mathbf{WCCT}$ to be *completely* characterised. The complete list of provided by Table 6 provides the *complete* list of the task management and administration service overheads incurred by tasks under the conditions assumed for DMS and ADS analysis. General task's execution overheads may obviously include those resulting from the server call services listed in table 3. The individual services in both table are fully described by semantically and operationally correct composition of their elementary constituents.

Table 6: *Tasking Overhead Bounds*

task / event	analysis case	
	DMS & ADS ($q = 0$)	**ADS** ($q > 0$)
Cyclic		
on release	Context_Switch + Delay_Until(Exit)	0
on suspension	Delay_Until(Enter) + Sched_Select + + Context_Switch	0
Int Sporadic		
on release	Interrupt_Handling + Sched_Select + Context_Switch + Interrupt_Sporadic_Wait(Exit)	
on suspension	Interrupt_Sporadic_Wait(Enter) + Sched_Select + Context_Switch	
Sw Sporadic		
on release	SPO.Wait(dequeue)	0 *if Releasing* $\epsilon HP(SS)$; SPO.Wait(dequeue) *otherwise*
on suspension	SPO.Wait(enqueue)	SPO.Wait(open) *if Releasing* $\epsilon HP(SS)$; SPO.Wait(enqueue) *otherwise*
Any Task Type		
RPO.Service	PO_Entry + $\mathbf{WCCT}$(Service) + PO_Exit	
SPO.Signal	SPO.Signal(no_wait) *if Caller* $\epsilon HP(SS)$; SPO.Signal *otherwise*	
Interval Timer		
on cyclic release	$\mathbf{K}_i^t$ * (Timer_Interrupt + Ready_After_Delay + Sched_Select)	

4 Conclusions and Outlook

This paper has presented the design and implementation of an Ada programming model intended for use on board of new-generation software-intensive satellite control systems. The programming model is based on an educated and optimised use of Ada tasking and preemptive priority-based scheduling. Preliminary analyses have shown that fixed-priority process-based preemptive scheduling suits the emerging application needs better than the conventional forms of rigid and inflexible cyclic scheduling. Acceptance of the novel approach, however, critically depends upon the provision of creditable means to statically ascertain the run-time performance of the system and its ability to meet the designated deadlines.

The choice made as part of the ERC32 model's design was to provide comprehensive support for worst-case response time analysis. This form of analysis, however, may easily incur excessive pessimism and consequently yield too conservative, low-efficiency predictions. This paper has described the approach taken to maximise accuracy of prediction and control of pessimism in the implementation of the ERC32 analysis model.

Acknowledgements

The work described in this paper is the result of the collective effort of all the members of the ERC32 project software team. The author is particularly indebted to Claire Bawin, Brian Dobbing, John Reynolds and Andy Walter for their great dedication and highly constructive contribution to the project.

References

[1] M. Joseph, P. Pandia. "Finding Response Times in a Real-Time System". *The Computer Journal* (BCS), 29(5), pp 390-395. 1986.

[2] International Standard ISO/IEC 8652:1987. *Ada Reference Manual*. September 1987.

[3] J.B. Goodenough, L. Sha. "The Priority Ceiling Protocol: A Method for Minimising the Blocking of High-Priority Tasks". *Ada Letters*, 8(7), pp 35-38. 1988.

[4] J.P. Lehoczky, L. Sha, Y. Ding. "The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior". *Proc. 10th Real-Time Systems Symposium*, pp 166-171. IEEE. December 1989.

[5] L. Sha, R. Rajkumar, J.P. Lehoczky. "Priority Inheritance Protocols: An Approach to Real-Time Synchronization". *Transactions on Computers*, 39(9), pp 1175-1185. IEEE. September 1990.

[6] J.A. Stankovic, K. Ramamritham. "What is Predictability for Real-Time Systems?". *Real-Time Systems*, (2), pp 247-254. 1990.

[7] N.C. Audsley, A. Burns, M.F. Richardson, A. Wellings. "Hard Real-Time Scheduling: The Deadline Monotonic Approach". *Proc. 8th Real-Time Operating Systems and Software*, pp 127-132. IEEE. 1991.

[8] C.D. Locke. "Software Architectures for Hard Real-Time Applications". *Real-Time Systems*, (4), pp 37-53. 1992.

[9] J. Roos, F. Gomez-Molinero. "A Complete Version of the Ada Tasking Coprocessor". *Proc. Real-Time Embedded Processing for Space Applications*. CNES. November 1992.

[10] N.C. Audsley, A. Burns, A. Wellings. "Deadline Monotonic Scheduling Theory and Application". *Control Engineering Practice*, 1(1), pp 71-78. 1993.

[11] British Aerospace Plc. "Hard Real-Time Operating System Kernel". *Final Report TP1079 of ESTEC Study Contract No. 9198/90/NL/SF*. February 1993.

[12] Vega Space Systems Engineering Ltd. "Study of the Operations Aspects of On-Board Software". *Final Report on ESTEC Purchase Order No. 123509*. June 1993.

[13] C. Bailey, E. Fyfe, T. Vardanega, A. Wellings. "The Use of Pre-Emptive Priority Based Scheduling for Space Applications". *Proc. 14th Real-Time Systems Symposium*, pp. 253-257. IEEE. December 1993.

[14] T. Vardanega. "Experience with the Development of Hard Real-Time Embedded Ada Software". *Proc. 16th Int. Conf. on Software Engineering*, pp. 301-308. IEEE. May 1994.

[15] A. Burns, A. Wellings. "HRT-HOOD: A Structured Design Method for Hard Real-Time Systems". *Real-Time Systems*, (6), pp 73-114. 1994.

[16] J. Gaisler. "Concurrent Error-detection and Modular Fault Tolerance in a 32-bit Processing Core for Embedded Space Flight Applications". *Proc. FTCS-24*, pp. 128-130. IEEE. June 1994.

[17] K.W. Tindell, A. Burns, A. Wellings. "An Extendible Approach for Analysing Fixed Priority Hard Real- Time Tasks". *Real-Time Systems*, (6), pp 133-152. 1994.

[18] International Standard ISO/IEC 8652:1995. *Ada Reference Manual*. February 1995.

[19] T. Vardanega. "On the Use of Ada Tasking in the Building of Satellite Control Software". *Proc. 3rd Workshop on Parallel and Distributed Real-Time Systems*, pp. 44-51. IEEE. April 1995,

[20] Intecs Sistemi SpA. "HRT-HoodNICE: A Hard Real-Time Software Design Support Tool". *Final Report on ESTEC Contract No. 11234/94/NL/FM(SC)*. March 1996.