The Ravenscar Tasking Profile for High Integrity Real-Time Programs

Brian Dobbing
Aonix-Europe Ltd
Partridge House, Newtown Road
Henley-on-Thames RG9 1EN UK
+44 1491 415016
brian@uk.aonix.com

Alan Burns
University of York
Heslington
York, UK
burns@minster.cs.york.ac.uk

1. ABSTRACT

The Ravenscar Profile defines a simple subset of the tasking features of Ada, in order to support efficient, high integrity applications that need to be analyzed for their timing properties. This paper describes the Profile, which is being endorsed by the ISO working group that is giving guidelines on the use of Ada in high integrity systems. An implementation of the Profile is then described in terms of development practice, run-time characteristics, certification, size and performance. The important issue of enforcing the restrictions imposed by the Ravenscar Profile at compile-time using Ada95 pragma Restrictions is also addressed.

1.1 Keywords

Ada, tasking, safety-critical, high-integrity, Ravenscar

2. INTRODUCTION

High-integrity systems traditionally do not make use of high-level language features such as Ada tasking. This is despite the fact that such systems are inherently concurrent. Concurrency is viewed as a 'systems' issue. It is visible during design and in the construction of the cyclic executive that implements the separate code fragments, but it is not addressed within the software production phases.

Notwithstanding this approach, the existence of an extensive range of concurrency features within Ada does allow concurrency to be expressed at the language level with the resulting benefits of having a standard approach that can be analyzed and checked by the compiler, and supported by other tools.

The requirement to analyze both the functional and temporal behavior of high integrity systems imposes a number of restrictions on the concurrency model that can be employed. These restrictions then impact on the language features that are needed to support the model. Typical features of the concurrency model are as follows.

- A fixed number of activities (we shall use the Ada term *task* to denote an independent concurrent activity).
- Each task has a single invocation event, but has a potentially unbounded number of invocations. The invocation event can either be temporal (for a time-triggered task) or a signal from either another task or the environment. A high-integrity application may restrict itself to only time-triggered tasks.
- Tasks only interact via the use of shared data. Updates to any shared data must be atomic.

These constraints furnish a model that can be implemented using fixed priority scheduling (either preemptive or non-preemptive) and analyzed in a number of ways:

- The functional behavior of each task can be verified using the techniques appropriate for sequential code (e.g. [1]). Shared data is viewed as just environmental input when analyzing a task. Timing analysis can ensure that such data is appropriately initialized and temporally valid.
- Following the assignment of temporal attributes to each task (period, deadline, priority etc), the system-wide timing behaviour can be verified using the standard techniques in fixed priority analysis (e.g. [2]).

3. TASKING FEATURES

The Ada95 language revision has both increased the complexity of the tasking features and provided the means by which subsets (or profiles) of these features can be defined. To all of the Ada83 features (dynamic task creation, rendezvous, abort) has been added protected objects, ATC (asynchronous transfer of control), task attributes, finalization, requeue, dynamic priorities and various low-level synchronization mechanisms. Subsets are facilitated by pragma `Restrictions` that allows various aspects of the language to be limited in scope or removed from the programmer completely.

Whilst the full language produces an extensive collection of programming aids (e.g. see [3]) from which higher-level abstractions can be constructed, there are a number of motivations for defining restricted models:

- increasing efficiency by removing features with high overheads
- eliminating non-deterministic features for safety-critical applications
- simplifying the run-time kernel for high-integrity applications
- removing features that lack a formal underpinning
- removing features that inhibit effective timing analysis

Of course the necessary restrictions are not confined to the tasking model, but this paper only considers concurrency. To implement a restricted concurrency model in Ada requires only a small selection of the available tasking features. At the Eighth International Real-Time Ada Workshop (1997) the following tasking subset (called the *Ravenscar Profile*) was defined for high-integrity, efficient, real-time systems [4]. This work was input to a study carried out by the International Standards Organization Working Group 9, Annex H Rapporteur Group, that led to the production of a report giving guidelines on the use of Ada in high integrity systems. (This report is currently at the draft stage and is not yet publicly available.)

4. RAVENSCAR CONCURRENCY MODEL

4.1 Decomposition

The application should be decomposed into a number of separate processes, each with a single thread of control, with all interaction between these processes identified. This decomposition is normally the result of applying a design methodology suitable to describe real-time systems, such as the Unified Modeling Language (UML) [5].

Each process is implemented as a separate Ada task. The tasks are categorized as *cyclic* (meaning that they execute periodically using a statically-assigned rate), or *event-driven* (meaning that they execute in response to an asynchronous event). In addition, protected objects are used to provide mutually-exclusive access to shared resources (e.g., for concurrent global data access) and to implement task invocation via a single protected entry per object.

4.2 Ada Construct Restrictions

Tasks In order to be suitable for schedulability analysis, the task set to be analyzed must be static. This is reflected in the Profile restrictions that all Ada tasks and protected objects in the program are created at the outermost *library-level* and the tasks never terminate. The Profile does not support dynamic allocation of tasks or protected objects, nor dynamic alteration of task priorities via package `Ada.Dynamic_Priorities`.

Invocation events A cyclic (or "time-triggered") task invocation event is programmed using the *delay until* statement with an argument of type `Ada.Real_Time.Time`. This provides the most precise release of the task since the Time type is mapped onto the underlying system clock. Neither package `Ada.Calendar` nor the *delay_relative* statement is supported by the Profile.

An event-driven task invocation event is programmed using either a *Suspension Object* being set to True, or a *barrier expression* in a protected object entry body becoming True. The Profile simplifies the implementation of barrier expressions by restricting them to being either a Boolean variable that is part of the local protected state or a Boolean literal value. The Boolean variable can be set either by another task or by an interrupt handler.

General Execution In order to meet the deterministic execution profile requirement necessary for static analysis of the source code and for schedulability analysis, a number of tasking constructs are restricted by the Profile. In particular:

- *No use of task entries* – the Ada rendezvous can exhibit unbounded priority inversion and is not needed when all communication and synchronization can be achieved using protected objects.
- *No protected entry queues* – the non-determinism and runtime complexity associated with entry queues is eliminated in the Profile by requiring a maximum of one entry caller at any given time.
- *No entry selection policy* – the non-determinism and runtime complexity associated with selecting the next task to execute a protected operation as part of entry queue servicing is eliminated in the Profile by requiring a maximum of one entry per protected object.

Runtime Simplification A number of tasking constructs are restricted by the Profile to simplify the runtime system, making it smaller, faster and easier to certify. In particular:

- *No requeue* – the *requeue* statement has significant overhead and is difficult to analyze statically
- *No abort or asynchronous transfer of control* – these features lead to pervasive distributed overhead throughout the runtime system to cater for asynchronous task actions
- *No use of the select statement* – eliminates rarely used timed and conditional protected entry calls (in addition to complex rendezvous and asynchronous transfer of control)
- *No user-defined task attributes* – a dynamic feature that has runtime complexity and overhead.

4.3 Code Templates

The profile does not require the application to use any particular coding style for the execution of the tasks, protected objects, and interrupt handlers. However if the application is required to undergo schedulability analysis, certain task templates and coding styles are useful in defining the activities that are to be analyzed. These are described below:

Cyclic Task The task body for a cyclic task typically has, as its last statement, an outermost infinite loop containing one or more *delay until* statements [RM section 9.6]. (The basic form of a cyclic task has just a single delay until statement either at the start or at the end of the statements within the loop.) The model supports only one time type for use as the argument - `Ada.Real_Time.Time` [RM section D.8] - which maps directly to the underlying system clock for the maximum precision**.** Note that task termination is a bounded error condition in the Ravenscar profile; hence the loop is infinite. Example:

```
task body Cyclic is
  Next_Period : Ada.Real_Time.Time :=
     First_Release;

  Period : Ada.Real_Time.Time_Span :=
     Ada.Real_Time.Milliseconds(50);
  -- other declarations
begin
  -- Initialization code
  loop
    delay until Next_Period;
    -- Periodic response code
    Next_Period := Next_Period + Period;
  end loop;
end Cyclic;
```

Event-Driven Task The task body for an event-driven task typically has, as its last statement, an outermost infinite loop containing as the first statement either a call to an Ada protected entry [RM section 9.5] or a call to wait for the state of an Ada "Suspension Object" [RM section D.10] to become true.

The suspension object is the optimized form for a simple suspend/resume operation.

The protected entry is used when extra operations are required:

a. Data can be transferred from signaler to waiter atomically (i.e., without risk of race condition) by use of parameters of the protected operations and extra protected data.

b. Additional code can be executed atomically as part of signaling by use of the bodies of the protected operations. Example:

```
protected Event is
   entry     Wait   (D : out Data);
   procedure Signal (D : in Data);
private
   Current : Data; -- Event data decl'n
   Signalled : Boolean := False;
end Event;

protected body Event is
   entry Wait (D : out Data)
      when Signalled is
   begin
      D := Current;
      Signalled := False;
   end Wait;

   procedure Signal (D : in Data) is
   begin
      Current := D;
      Signalled := True;
   end Signal;
end Event;

task body Sporadic is
   My_Data : Data;
begin
   -- Initialization code
   loop
     Event.Wait (D => My_Data);
     -- Response code processing My_Data
   end loop;
end Sporadic;
```

Interrupt Handlers The code of an interrupt handler will often be used to trigger a response in an event-driven task. This is because the code in the handler itself executes at the hardware interrupt level, and so typically the major part of the processing of the response to the interrupt is moved into the task, which executes at a software priority level with interrupts fully enabled. The interrupt handler typically will store any interrupt data in its protected object and then releases the waiting event-driven task by changing the state of the protected data Boolean used as the entry barrier in the same protected object, as shown in the example protected object `Event`.

Shared Resource Protected Object A protected object used to ensure mutually exclusive access to a shared resource, such as global data, typically contains only protected subprograms as operations, i.e., no protected entries. Protected entries are used for task synchronization purposes. A protected procedure is used when the internal state of the protected data must be altered, and a protected function is used for information retrieval from the protected data when the data remains unchanged. Example:

```
protected Shared_Data is
  function  Get return Data;
  procedure Put (D : in Data);
private
  Current : Data;
    -- Protected shared data declaration
end Shared_Data;

protected body Shared_Data is
  function Get return Data is
  begin
    return Current;
  end Get;

  procedure Put (D : in Data) is
  begin
    Current := D;
  end Put;
end Shared_Data;
```

5. IMPLEMENTING THE PROFILE

Ada compiler vendor Aonix has undertaken the development of an Ada95 compilation system which implements the Ravenscar profile, known as *Raven* [6], hosted on Windows NT and Sparc Solaris, and targeting the PowerPC, 68K and Intel range of processors. This section describes some of the key elements of this implementation.

5.1 Development Practices

The principle goal of the implementation was to develop a runtime system for Ada95 restricted as per the Ravenscar profile, which was suitable for inclusion in:

- A safety-critical application requiring formal certification
- A high-integrity system requiring functional determinism and reliability
- A concurrent real-time system with timing deadlines requiring temporal determinism, eg. schedulability analysis
- A real-time system with execution time constraints requiring high performance
- A real-time system with memory constraints requiring small and deterministic memory usage

Consequently a rigorous set of development practices was enforced based on the traditional software development model, including:

- Documentation of the software requirements
- Definition and documentation of the design to meet these requirements, including traceability
- Formal design reviews
- Formal code walk-throughs of the runtime implementation
- Definition and documentation of the runtime tests to verify that the implementation of the design meets its formal specification
- Documentation of the formal verification test results
- Capture of all significant items within a configuration management system

5.2 Requirements

The software requirements include the following elements:

- The runtime design shall support both the preemptive and non-preemptive implementations of the Ravenscar profile.
- The runtime design shall optimize a purely sequential (non-tasking) program by not including any runtime overhead for tasking.
- The design shall structure the runtime such that a library of additional runtime Ada packages which have not undergone formal certification can be supplied as a stand-alone “extra”, for applications which require the extra functionality but not the rigors of certification.
- The runtime algorithms shall be coded such that the worst case execution time is deterministic and as short as possible
- The runtime algorithms shall be coded such that the average case execution time is as short as possible
- The runtime algorithms shall be coded so as to miminize the use of global data, and so as not to acquire memory dynamically. (The total global memory requirement of the runtime system shall be small and deterministic.)
- The runtime algorithms shall be coded so as to conform to the certification coding standards
- The runtime algorithms shall be coded so as to conform to the Ravenscar profile plus the implementation-defined sequential code restrictions
- A coverage analysis tool shall be provided for certification purposes
- A schedulability analyzer shall be provided which implements standard algorithms used in fixed-priority timing analysis.

- Enforcement of the Ravenscar profile, plus other restrictions on sequential constructs, shall be performed at compile-time wherever possible. (This eliminates runtime code to perform the checks, and the risk of runtime exceptions being raised in the event of check failure.)
- The runtime kernel shall be verified using the verification tests written to validate the correct implementation of the requirements.

5.3 Design Considerations

Restriction Enforcement This is achieved at compile time using the Ada pragma `Restrictions` [RM section 13.12]. Some new restriction identifiers were defined to cover restrictions not considered by the Ada standard. These new identifiers are currently being considered for standardization by the International Standards Organization Working Group responsible for the Safety and Security annex of the Ada standard. A file containing the set of pragma Restrictions required by the Profile is automatically compiled into the program library upon its creation.

Certifiability The requirements for certifiability of the runtime code using standards such as DO-178B [7] Level A impinge also on the source code itself (as well as its development process). Each file contains a header including:

- Overview of purpose or functionality
- Requirement(s) which are met
- Detailed definition of global data/parameter usage
- Detailed definition of algorithm

This description is checked against the actual code during walk-through audits, and is used to verify that the implementation conforms to the design, and that the design fully meets the requirements.

Worst Case Execution Time In order to perform accurate schedulability analysis, it is necessary to input the runtime execution overhead (see [8]). For hard real-time systems, in which the failure to meet a hard timing deadline is catastrophic to the entire system, worst case execution times are generally used in the computations.

For the application code itself, the user can obtain worst case execution times either by analyzing the Ada code (e.g. [9]) or measuring the times using tests that exercise the various code paths. But for the runtime system operations, the user has no direct way of knowing which scenario will produce the worst case time, unless the runtime source code is available and also documentation to describe the criteria which determine the execution path at each decision point. Thus for every runtime operation supported by the model, the vendor must provide "metrics" [RM section D (2)] which define its worst case execution time. For the runtime tasking kernel implementing the Ravenscar profile, this set of metrics will include:

- Overhead for protected operations, with and without a protected entry
- Overhead for processing of the **delay until** statement
- Overhead in handling timer interrupt and user interrupts
- Rescheduling overhead, including the time to perform a context switch

Clearly, this imposes strict constraints on the algorithms used to implement these operations such that their worst case execution time is not overly excessive. For example, use of a linear search proportional to the maximum number of tasks in the program would be unacceptable for a program with a large number of tasks. So, the runtime contains optimizations to minimize critical worst case timings.

Performance Several techniques are used to improve the performance of the runtime. Simple and very short runtime subprograms can be defined as having calling convention "Intrinsic" [RM section 6.3.1], which means that their code is built into the compiler and is used directly in place of the call. Typically this is used for immutable code sequences such as arithmetic and relational operators for the Time type [RM section D.8] and for highly time critical simple operations such as getting the identity of the currently-executing task.

Other short subprograms can be defined as being "inlined" [RM section 6.3.2], which gives similar performance gain by avoiding the procedure call and return overhead, but without having to actually build the generated assembler code into the compiler code generator.

Early indications of the performance of the runtime system are very encouraging and are listed in Table 1.

(Microseconds)	Rendezvous (VxWorks)	Protected Object (VxWorks)	Protected Object (Ravenscar)
PIWG T000001	375.0	12.9	1.5
PIWG T000002	395.8	12.9	1.5

Table 1 – Runtime Performance on Ultra 604 133MHz

(K-bytes)	Code	Data	Stack
Null program	3.7	0.45	0.81
Hello World	4.2	0.52	0.86
Minimal Tasking	12.1	1.10	1.80

Table 2 – Application Program Sizes (Power PC)

Runtime Size The runtime was designed and coded to minimize the size of both the code and the data. For example, an important optimization in the Ada pre-linker tool (the "binder") is elimination of uncalled subprograms from the executable image. But this optimization is only fully effective if the code is structured in a very modular way. For example, there is the requirement that no runtime code or data which is specific to Ada tasking should be included in the executable if the program does not use tasking.

The major component of the runtime data is the stack and Task Control Block (TCB) which is required for each task's execution. Each application program is required to declare the memory areas to be used for the stacks and TCBs in the Board Support Package. This provides a simple interface to tune the stack sizes to the worst case values, whilst also giving full application-level determinism on the amount of storage which is reserved for this purpose. Early indications of the size of the runtime system are very encouraging and are listed in Table 2.

6. CONCLUSION

This paper has described the Ravenscar profile, a subset of Ada95 tasking intended to model concurrency in safety-critical, high-integrity, and general real-time systems. The profile is included in guidelines produced by the ISO WG9 Annex H Rapporteur Group on the use of Ada in high integrity systems. The use of a powerful, structured and highly-checked language such as Ada is vitally important in all market sectors demanding high reliability and efficiency.

The paper has also described a commercial-off-the-shelf implementation of the profile for the PowerPC processor family which has proved the feasibility of developing production-quality tool support and a certification-quality runtime system for the Ravenscar profile.

7. REFERENCES

[1] Barnes, J. High Integrity Ada – The SPARK Examiner Approach, Addison Wesley Longman Ltd (1997).

[2] Klein, M.H. et al, A Practioner's Handbook for Real-Time Analysis : A Guide to Rate Monotonic Analysis for Real-Time Systems, Kluwer Academic Publishers (1993)

[3] Burns, A. and Wellings, A.J. Concurrency in Ada, Cambridge University Press (1995)

[4] ACM Ada Letters, Proceedings of the 8th International Real-Time Ada Workshop: Tasking Profiles (September 1997).

[5] Object Management Group, UML Specification ad/97-08-02 thru ad/97-08-11 http://www.omg.org/library/schedule/Technology_Adoptions.htm#tbl_UML_Specification (November 1997).

[6] Aonix Inc, ObjectAda Real-Time Windows NT x PowerPC/Raven (1998)

[7] RTCA Inc, Software Considerations in Airborne Systems and Equipment Certification, RTCA/DO-178B/ED-12B. (December 1992)

[8] Katcher, D. et al, Engineering and Analysis of Fixed Priority Schedulers, in IEEE Trans. Software Engineering 19 (1993)

[9] Chapman, R., Burns, A., and Wellings A.J., Combining Static Worst-Case Timing Analysis and Program Proof in Real-Time Systems 11(2):145-171 (September 1996)

[10] and [RM] Intermetrics Inc. Ada95 Reference Manual, ANSI/ISO/IEC-8652:1995, (January 1995)