

## High Integrity Ravenscar

Peter Amey and Brian Dobbing

Praxis Critical Systems, 20, Manvers St., Bath BA1 1PX, UK  
peter.amey@praxis-cs.co.uk  
brian.dobbing@praxis-cs.co.uk

**Abstract.** The Ravenscar Profile is an exciting development for the Ada community since it provides, for the first time in the history of our industry, support for deterministic, multi-tasking programming as an integral part of a standardized language. Despite its many advantages, the profile leaves several areas where behaviour is implementation defined and can result in run-time errors; this is unfortunate in a profile aimed clearly at the critical systems market. The SPARK language is a well-established sequential Ada subset that avoids ambiguity and allows all language rule violations to be detected prior to execution. The authors show how the principles of SPARK have been successfully extended to encompass the Ravenscar Profile thereby statically eliminating the profile's problematic areas. The result should allow concurrent Ada programs to be constructed with the same degree of rigour that is now possible using sequential SPARK.

### 1 Introduction

The Ravenscar Profile (the definition of which can be found in Section 3 of [1]) is now established as providing the basic building blocks for constructing high integrity Ada95 concurrent programs. The profile has been accepted for inclusion in the next revision of the Ada language standard and is already supported by the major Ada product vendors. In addition, specialized versions of Ada runtime systems that implement the profile's concurrency model, whilst excluding the other concurrency features of Ada95 that are restricted by the profile definition, have been developed. In some cases, these Ada runtime systems are supported by certification evidence for rigorous standards such as RTCA/DO-178B level A [2].

This support for the profile forms an acceptable baseline for meeting the requirements of the dynamic semantics of high integrity systems. However, this basic level of support needs to be supplemented by additional rules and by static analysis techniques in order to be able to show the same level of proof of correctness and absence of run-time errors that is currently achievable in sequential programs using tools such as the SPARK Examiner [3, 4]. The kinds of problems that need to be addressed and the role of static analysis techniques, are described in section 6.2 of [1]. Essentially this identifies two roles for analysis:

1. ensuring that a program is well-formed; and free from run-time errors and erroneous behaviour; and
2. providing evidence for its overall correct behaviour.

The paper is in three main sections: section 2 briefly restates the problems of well-formedness introduced by the Ravenscar Profile. Section 3 shows how extensions to the SPARK language and SPARK Examiner are sufficient to detect all such problems, statically, before execution. Finally, section 4 outlines further forms of analysis contributing to the second of the two roles enumerated above.

## 2 Problems Associated with the Ravenscar Profile

Although it represents an enormous step forward in its support for reliable concurrent programming, the Profile identifies a number of error conditions which may give rise to run-time exceptions, erroneous behaviour or implementation-defined behaviour. Section 6 of [1] provides a detailed explanation of these problem areas and outlines theoretical approaches that could be taken to detect and eliminate them statically. We briefly restate the problems here before describing our practical analysis system for their elimination.

### 2.1 Errors Leading to Run-time Exceptions

**Effect of Unexpected Exceptions** The general concern within high integrity systems of the occurrence of unhandled exceptions is not addressed by the Ravenscar Profile since exceptions relate to the sequential, rather than the concurrent, part of the language. Nevertheless, whereas an unhandled exception will cause a sequential program to terminate, and hence offer an immediate opportunity for some program level control to invoke recovery actions, an unhandled exception during the execution phase of a concurrent program is not so readily detected. In particular, an unhandled exception can cause any of the following effects:

1. silent abandonment of the execution of an interrupt handler;
2. silent termination of a task;
3. premature exit from a protected action, possibly leaving it in an inconsistent state.

Appropriate static analysis techniques already exist to show proof of absence of run-time errors in sequential code due to language-defined exceptions, see for example [5]. The same techniques can also be applied to the sequential code within each task and protected object, leaving only the possibility of the exceptions that relate directly to the concurrency behaviour. These techniques can therefore be used to show statically that all three of the above effects of an unhandled exception due to check failure cannot occur. Since the elimination of run-time errors in this way is not specific to the Ravenscar Profile, it is not considered further in this paper.

**Exceptions Due to Concurrency** Of the concurrency checks defined by Ada95, there are only two that apply to a Ravenscar Profile program:

1. detection of priority ceiling violation as defined by the Ceiling Locking policy — calls from a task to protected operations must follow a non-decreasing priority chain;

2. detection of violation of not more than one task waiting concurrently on a suspension object (via the `Suspend_Until_True` operation).

In addition, two further concurrency checks are introduced by the Ravenscar Profile definition:

1. the maximum number of calls that are queued concurrently on a protected entry shall not exceed one;
2. a potentially blocking operation shall not be executed by a protected action.

## 2.2 Errors Leading to Erroneous Behaviour

**Use of Unprotected Shared Variables** If two tasks share an unprotected variable the resulting program may be erroneous. In principle this does not prevent the sharing of such variables; however this would require a demonstration that the temporal properties of the program prevented concurrent access to it. More robust protection requires a check that unprotected data *can never* be shared.

**Race Conditions During Elaboration** Within a sequential program, detection of access before elaboration errors is generally straightforward during program development due to the repeatable nature of the elaboration order, and the raising of Program Error exception at the point of failure, causing the program to terminate. Undesired elaboration order variation can be prevented by explicit use of elaboration order pragmas. SPARK is even stronger in this respect since it eliminates all dependency on elaboration ordering.

Section 6.2.2 of [1] provides a detailed explanation of why the situation is less clear cut in the case of concurrent programs and, in particular, the risk of race conditions during elaboration. These problems can be eliminated by the use of a new `Partition_Elaboration_Policy` pragma which has been agreed for addition to the next revision of the Ada language standard; however, this pragma is *not* part of the Ravenscar Profile and not yet part of the Ada language. SPARK takes a different approach to elaboration order problems as described in Section 3.4.

**Program Incompleteness** It is not easy to ensure that all the tasks and protected objects required to provide a program's intended behaviour have actually been included in its executable image. Unlike sequential code, the failure to "with" an active component does not make the program illegal; it simply becomes a legal but different program performing a subset of its intended action.

## 2.3 Errors Leading to Implementation-defined Behaviour

**Task Termination** The Ravenscar Profile includes the `Restrictions` pragma `No_Task_Termination`, but the dynamic effect of such termination is implementation-defined. Attempts at adding a formal task termination handling mechanism to the next revision of the Ada language standard are at an immature state of development. A means of showing that tasks will *not* terminate is therefore essential if implementation-defined behaviour is to be avoided.

### 3 Static Elimination of Ravenscar Errors

The problems outlined in Section 2 can be addressed and eliminated by static analysis. In the case of SPARK we have extended the SPARK95 language and implemented additional checks within its supporting SPARK Examiner tool.

#### 3.1 Background

The rationale for SPARK, which is fully described in [3], places great emphasis on two things:

**Precision** SPARK is not about the detection and elimination of *some* errors, it is concerned with the exact representation of programs and the elimination of all ambiguous interpretations of them.

**A constructive approach** We believe in the maxim “Correctness by Construction”. It should be possible to check continuously, throughout development, that a program is progressing towards its planned goal.

These goals remain unchanged in the context of the current work and we have therefore had to devise language rules whose violation can be detected *under all circumstances* and to ensure that analysis can take place on incomplete programs, especially in the absence of all *bodies* of program units. The design approach follows the pattern used for the design of the sequential SPARK language. The required language properties are obtained by the combination of two tactics:

1. additional language restrictions policed by wellformation checks; and
2. the use of annotations to clarify the programmer’s intentions and to support the analysis of incomplete programs by asserting properties of units whose bodies may not yet be available for analysis.

#### 3.2 Additional Language Rules

Here we are concerned with the compilable core language of SPARK rather than with the definition of its annotations or language checks that make use of those annotations. Very few rules additional to those either already required by sequential SPARK or imposed by the Ravenscar Profile are required. The principal restrictions are as follows:

1. All task and protected *types* must be declared in package specifications. This is to facilitate the analysis of incomplete programs during development. Note that SPARK does not require task and protected *objects* to be placed in specifications although, in accordance with the Ravenscar rules, they must be declared at library level.
2. Discriminants of task and protected types must be static and, currently, may not include access discriminants. (We believe the latter restriction may be removable provided the former remains).
3. Protected elements must be initialized at declaration.

4. Each task must have a plain loop with no exits as its final statement. If the Environment Task is the only task, in a program that contains other concurrency features, e.g. an interrupt handler, then it must end with such a loop.
5. The attribute 'Count is not supported, nor is pragma Atomic .Components.

In addition, there are some naming restrictions for protected operations to prevent the need for overload resolution which is excluded from SPARK.

### 3.3 Additional Annotations

SPARK has an annotation at the package level which indicates that a package contains “state variables” and allows the effect of the package’s operations on that state to be described. This *own variable* annotation has been extended to allow the state to be identified as **protected** or as a **task**. It can also be followed by a *property list* which indicates such things as: priority, whether interrupts are involved and whether a task may suspend and on what variable(s). The property list is deliberately extensible and uses identifiers rather than new reserved words; this makes it feasible to extend the annotation system to support third party tools such as timing analysers and model checkers.

The property list may also be used as part of a procedure or task type annotation where it can be used to indicate, for example, that the procedure may delay and must be considered a potentially blocking operation.

Some properties take arguments or list of arguments in the form of named aggregates. The current list of properties and their meanings is as follows:

<i>Property</i>	<i>Location</i>	<i>Argument</i>	<i>Meaning</i>
<b>Priority</b>	own protected	expression	announces the priority that the object <i>will</i> have when it is declared
<b>Suspendable</b>	own protected	none	indicates that the object is a <i>predefined suspension object</i> or a protected object with an <i>entry</i>
<b>Interrupt</b>	own protected	optional, see example	indicates that the object contains one or more interrupt handlers and gives (optionally) a user-defined name for them
<b>Protects</b>	own protected	identifier list	shows that the unprotected variables in the identifier list are used solely by the protected object and can be treated exactly like protected elements
<b>Suspends</b>	unprotected procedure or task type	identifier list	indicates that the operation suspends on the identifiers given (by calling an entry in a named protected object, for example)
<b>Delay</b>	unprotected procedure	none	marks a procedure as potentially executing a delay statement (or other blocking action)

The validity of these claimed properties is, of course, checked when the body of the unit concerned is analysed. It is, for example, an error for a subprogram to execute a delay statement unless its specification annotation includes the delay property.

The manner in which SPARK eliminates the Ravenscar errors described above, and the additional analysis that can be achieved, is illustrated using a small example program which we describe now. The program provides a simple stopwatch: three user buttons allow the stopwatch to be started, stopped and reset; these are achieved by interrupt routines, attached to each button, that set or reset a suspension object that controls the main timing loop. The timing loop is a periodic task; when released, this cycles at 1 second intervals and calls a protected object which is responsible for maintaining the current count of seconds and passing it to an out port that causes it to be displayed. The reset button clears the time count to zero but does not start or stop the timing loop.

In our illustrative example, we have three packages: *User* which provides the control buttons; *Timer* which contains the task providing the main timing loop; and *Display* which maintains the second count and copies it to the display port each time it changes. Applying the above annotations to these package specifications we have:

```

package User
--# own protected Buttons : PT (Interrupt =>
--#                               (StartClock => StartButton,
--#                               StopClock  => StopButton,
--#                               ResetClock => ResetButton),
--#                               Priority => 10);
is
  protected type PT is
    pragma Interrupt_Priority (10);

    procedure StartClock;
    --# global in out Timer.Operate;
    --# derives Timer.Operate from Timer.Operate;
    pragma Attach_Handler (StartClock, 1);

    procedure StopClock;
    --# global in out Timer.Operate;
    --# derives Timer.Operate from Timer.Operate;
    pragma Attach_Handler (StopClock, 2);

    procedure ResetClock;
    --# global in out Display.State;
    --# derives Display.State from Display.State;
    pragma Attach_Handler (ResetClock, 3);
  end PT;
end User;

```

This tells us that the package will contain a protected own variable called `Buttons`, of type `PT`. This object provides interrupt handling and, optionally in this case, we have chosen to associate each of 3 interrupt handling procedures with a programmer-selected name that will be used in the partition wide flow analysis (see Section 4.1 for an explanation of how these names are used). Finally, the priority of the object is announced. The SPARK Examiner can be made aware of the definition of the implementation-dependent subtypes for priority ranges and will make appropriate range checks on the values used in the annotation.

```

package Display
--# own out      Port;
--#      protected State : PT (priority => 10, protects => Port);
is
  procedure Initialize;
  --# global in out State;
  --# derives State from State;

  procedure AddSecond;
  --# global in out State;
  --# derives State from State;

  protected type PT is
    pragma Priority (10);

    procedure Increment; -- add 1 second to stored time and send it to port
    --# global in out PT; --note use of type name here means "myself" or "this"
    --# derives PT from PT;

    procedure Reset; -- clear time to 0 and send it to port
    --# global in out PT;
    --# derives PT from PT;

  private
    Counter : Natural := 0;
  end PT;
end Display;

```

This tells us that the package contains an own variable called `Port` which is an out port (this is a conventional, sequential SPARK annotation) and a protected own variable `State` of priority 10 which *owns and controls* the port. The latter information is very useful; protected elements cannot include external objects such as those with associated address clauses or `pragma Import`, so great care is needed to ensure that concurrent access to a shared port cannot occur. The *protects* property may only be used to provide protection for an otherwise unprotected own variable declared in the same package as the protected object that will protect it; the SPARK Examiner then ensures by static semantic checks that the port is never accessed from outside the protected object that protects it. Finally, we consider the main timing package:

```

package Timer
--# own protected Operate (suspendable);
--#      task TimingLoop : TT;
is
  -- These two procedures simply toggle suspension object Operate
  procedure StartClock;
  --# global in out Operate;
  --# derives Operate from Operate;

  procedure StopClock;
  --# global in out Operate;
  --# derives Operate from Operate;

```

```

task type TT
--# global in out Operate, Display.State;
--#      in      Ada.Real_Time.ClockTime;
--# derives Operate      from Operate &
--#      Display.State from Display.State &
--#      null          from Ada.Real_Time.ClockTime;
--# declare suspends => Operate;
is
      pragma Priority (10);
end TT;
end Timer;

```

Note that the task type includes a *declare* annotation which states that tasks of type TT may perform a suspension operation on object Operate.

The combination of the information provided in these annotations and the SPARK rule requiring task and protected *types* to be declared in package specifications means that, without access to package bodies, we have sufficient information to eliminate the Ravenscar errors outlined in Section 2. Some errors are detected during the examination of individual program units such as a package body and others when the main program (more correctly *Environment Task*) is analysed. Note that the latter class of check requires access only to the package specifications “withed” by the main program; at no stage is a complete, linkable closure of the entire program required.

### 3.4 Static Elimination of Ravenscar Errors – A Practical Implementation

**Priority Ceiling Violation** In order to ensure that the priority ceiling check cannot fail, we identify all protected objects that may be called directly or indirectly from each task (including the Environment Task) and each interrupt handler, via the *global* annotations. The transitivity rule for this annotation ensures that the identity of indirectly-called protected objects propagates to the root of the call tree. The priority of each protected object is available at specification level via its *property* annotation. The priority of each calling task is known from its subtype. Since all priorities in SPARK must be static (including via actual values of discriminants), we can ensure that each protected object call chain does not cause a priority ceiling violation.

**Protected Entry and Suspension Object Queue Violation** We enforce the protected entry and suspension object queue length check by ensuring statically that there can be at most one caller that can suspend on each object. Although this is more restrictive than is strictly required since queue length violations could be avoided by the temporal properties of the program, detection of timing behaviour is beyond the scope of the SPARK toolset. The restriction may be relaxed in the future if additional capability such as model checking is integrated. The implementation is based on the *suspends* property that identifies the objects that the caller may suspend on. This property is transitive, and so the list of all objects that each task may suspend on appears at the root of the call tree. The check cannot fail if the intersection of these lists is null.

**Execution of Potentially-Blocking Operation Violation** Detection of calls to potentially-blocking operations from a protected action is supported in SPARK using the transitive *delay* or *suspends* property of a procedure. If a protected body may execute a delay statement or protected entry call directly, or may call a procedure that has at least one of the *delay* or *suspends* properties, then an error is raised. As above, it may be possible to relax this rule in the presence of integrated model checking if this can show that no state exists for which the protected action does call the blocking operation.

**Use of Unprotected Shared Variables** In SPARK, we avoid the possibility of shared use of unprotected variables by allowing at most one task to access each unprotected global variable. We also prohibit protected objects from accessing unprotected state. These rules are enforced via the *global* annotations that identify the global variables that are referenced by each task, and *own* variable annotations that supply the protected property information for these variables. Note that a protected own variable includes a suspension object, an object of pragma Atomic type and objects protected via the *protects* property, in addition to Ada protected objects. As above, the rule may be relaxed in the future if model checking can show that concurrent access to an unprotected shared variable is not possible.

**Race Conditions During Elaboration** To avoid race conditions during elaboration we need to ensure that no task or interrupt handler is dependent, for its correct behaviour, on the earlier execution of package body elaboration code. Several SPARK rules and annotations combine to ensure that this is the case. We require that all global variables marked in their own variable annotation as *protected* are statically initialized:

- at declaration (for protected elements, protected variables of pragma Atomic types and non-external own variables protected via the *protects* property);
- automatically (as is the case for predefined suspension objects); or
- by the external environment (as is the case for a variable marked as an *in port* which is protected using the *protects* property).

We can therefore be sure that no protected state initialisation depends on body elaboration code. This leaves one further case to consider: the use of unprotected state by a single task. The model adopted is to ensure that the unprotected state is initialised only by the sole-user task, and not by library package elaboration code. This is enforced by examination of the task type's *global* and *derives* annotations to determine its use of unprotected variables, together with the *initializes* annotation for each of these variables, which defines whether the variable is initialised by elaboration code or not. Note that these rules do not apply to the Environment Task because the main subprogram is guaranteed not to run until all library package elaboration is complete; this is particularly useful since it provides full upward compatibility for existing legal sequential SPARK programs that may use initialised unprotected state.

We anticipate being able to relax the initialisation rules concerning use of unprotected state by tasks once pragma `Partition_Elaboration_Policy` is incorporated into the Ada language standard.

**Task Termination** Task termination is prevented by the language rule requiring each task to end with a plain loop with no exit statements and by the elimination of run-time exceptions from the program.

## 4 Program-wide Static Analysis

In addition to showing absence of run-time errors, it is also highly desirable to apply existing static analysis techniques for sequential code to an entire program that includes tasks, protected objects and interrupt handlers. Current technology supports data flow analysis, information flow analysis and proof that includes the use of pre and post conditions and assertions. For concurrent programs we also wish to support these analyses and to accommodate schedulability analysis and model checking.

### 4.1 Partition-Wide Information Flow Analysis

**Thread level analysis** Sequential SPARK's data and information flow analysis is described in [6]. At the thread level, where we are concerned with an individual task or subprogram, this is largely unaffected by the addition of concurrency constructs. In particular, we do not concern ourselves with temporal aspects or with the effect of task suspension. Therefore delaying or waiting on an entry or suspension object does not affect task-level flow analysis. In summary, flow analysis of tasks and interrupt handlers is performed on the basis that they *will be* activated at some stage.

The only real change to sequential flow analysis is that references to potentially shareable protected variables must be considered volatile at all times because the value read may be generated by another program thread at any time. Updating a potentially shared protected variable does not mean that the value written will still be there when the object is next referenced. For example, if we foolishly try to exchange the values of variables X and Y using protected variable P as a temporary store:

```
P := X; X := Y; Y := P; -- dangerous, P may no longer contain X
```

then we find that instead of being described by the following flow relation (as it would be if P was not shareable):

```
--# derives Y from X & X from Y & P from X;
```

we must write:

```
--# derives Y from X, P & X from Y & P from X;
```

which indicates that the final value of Y may depend not only on X but on any value of P that may be stored in the protected variable, by another thread, during its execution. This addition to the flow relation provides the necessary hook to allow us to track inter-task communication at the partition level.

**Partition-wide Information Flow Analysis** The SPARK annotation system is extended to include an additional *global* and *derives* annotation that describes the intended flow relation for the entire program partition. This intended behaviour is compared with a flow relation constructed as follows:

**For each task** (identified by the *own task* annotation of each package “withed” by the Environment Task), we add any object that the task suspends on (identified by the task’s property annotation) as an import that influences the exports of that task.

**For each interrupt handler** (identified by the property list of the *own protected* annotation of each “withed” package), we add, as an import, the name of the source from which the interrupt is deemed to come. By default this is the name of the protected object that contains the interrupt handler but the property annotation allows a more descriptive, user-selected name to be used instead (the use of names of *system* significance in annotations is encouraged, see [7]). The ability to name the source of interrupts is especially useful if a protected type declares more than one handler (as in our example) or if there are multiple objects of the same type.

**For the enriched annotations generated above** we take their union so as to establish connections between values generated by one task and referenced by another.

**Finally** we take the transitive closure of this union because each task runs continuously and the effects of inter-task information flows will eventually propagate to all tasks that share information. For example if Task 1 derives B from A and Task 2 derives C from B and Task 3 derives D from C then taking the closure ensures that the influence of A on D is detected.

It is important to note that none of the above steps require access to the bodies of the packages “withed” by the Environment Task. The information in the *own* variable annotations, associated property lists and the type declarations themselves is all that is needed. We can therefore check that our program is properly constructed before getting involved in implementation details. The veracity of the annotations is, of course, checked when the bodies are written.

We can obtain the partition flow relation for our stopwatch example by applying the above rules. The three interrupt handlers have their user-supplied names (from the *interrupt* property list) added as imports. The task has the suspension object it suspends on (from its *suspends* property list) added as an import. The relations are then unioned and closed giving:

```
--# derives Timer.Operate from Timer.Operate
--#           User.StartButton, User.StopButton &
--#           Display.State from Display.State
--#           Timer.Operate, User.StartButton,
--#           User.StopButton, User.ResetButton &
--#           null from
--#           Ada.Real_Time.ClockTime;
```

which provides the useful information that all three user buttons affect the displayed time but the reset button does *not* affect suspension object `Timer.Operate` which controls whether the clock is running.

As well as providing a useful description of overall system behaviour, the partition flow analysis provides protection from the particularly nasty error of program incompleteness (see section 2.2). With SPARK this error cannot occur because if an active program component is omitted then its effects will not be included in the calculated partition flow relation which is unlikely to agree with the claimed partition-level *global*

and *derives* annotation. The check ensures that the program elements actually included in the program closure are those that provide the claimed and expected information flow.

## 4.2 Timing Analysis and Model Checking

The work described in this paper and incorporated in the SPARK language and toolset dovetails nicely with other research work on the Ravenscar Profile. The steps described above ensure that a Ravenscar program is “well formed” and will not violate the specified rules of the profile; however, they are not sufficient to ensure that the program will meet its timing deadlines. Further work on the model checking and response time analysis aspects of the concurrency model is on-going and is the subject of other research papers that are being developed by the University of York. Eventually it is hoped that these two analysis streams will supplement one another (for example, by extending SPARK’s *property* annotation to provide information needed by timing analysis tools) to provide evidence of both statically and dynamically appropriate behaviour.

## 5 Conclusions

Ada remains unique in its comprehensive language-level support for multi-tasking. The Ravenscar Profile provides a framework for constructing dependable tasking programs with deterministic and analyseable timing properties. The extension of SPARK to encompass Ravenscar provides a way of statically showing such a program to be well-formed and free from run-time errors. The combination of these techniques with suitable, certifiable, run-time support must represent the most rigorous environment for producing high-integrity tasking programs.

## References

1. Burns; Dobbing; Vardanega: *Guide for the use of the Ada Ravenscar Profile in high integrity systems*. University of York technical report YCS 348 (2003)
2. *RTCA-EUROCAE: Software Considerations in Airborne Systems and Equipment Certification*. DO-178B/ED-12B. 1992.
3. Finnie, Gavin et al: *SPARK 95 - The SPADE Ada 95 Kernel — Edition 3.1*. 2002, Praxis Critical Systems<sup>1</sup>.
4. Barnes, John: *High Integrity Software - the SPARK Approach to Safety and Security*. Addison Wesley Longman, ISBN 0-321-13616-0. 2003
5. Chapman, Rod; Amey, Peter: *Industrial Strength Exception Freedom*. Proceedings of ACM SIGAda 2002<sup>2</sup>
6. Bergeretti and Carré: *Information-flow and data-flow analysis of while-programs*. ACM Transactions on Programming Languages and Systems 1985<sup>1</sup>, pp37-61.
7. Amey, Peter: *A Language for Systems not Just Software*. Proceedings of ACM SIGAda 2001<sup>2</sup>.

---

<sup>1</sup> Also available from Praxis Critical Systems

<sup>2</sup> Also downloadable from [www.sparkada.com](http://www.sparkada.com)