# Static Analysis of Ravenscar Programs

P.N. Amey and B.J. 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 provides support for deterministic, multi-tasking programming as an integral part of a standardized language. A key element in the exploitation of the advantages of the Profile within the critical systems market is its use with verification tools. An established static analysis tool for this market operates on sequential programs in the SPARK language, which is an annotated subset of Ada 95 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 Profile, thereby realising the benefits of constructing multi-tasking programs with the same degree of rigour that is currently 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 Ada 95 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 have been developed as COTS products. In some cases, these Ada run-time 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 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 static analysis:

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

This paper starts with a brief background section on the rationale behind the SPARK language, and how this has been preserved in its extension to support the profile. Two sections follow that address how each of the roles for static analysis has been addressed in the extensions to the SPARK language. There is then a brief example to illustrate the extensions, and finally there is a concluding section.

## 2 SPARK Rationale

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

1. 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.
2. SPARK is based on the maxim *Correctness by Construction*. It should be possible to check continuously, throughout development, that a program is progressing towards its planned goal.

These principles provide an exact match with the roles that have been identified above for static analysis. The support for the Ravenscar Profile has thus devised language rules whose violation can be detected *under all circumstances*, and has ensured that program-wide analysis of correctness 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 well-formation checks; and

2. the use of annotations to define the programmer's intent and to support the analysis of incomplete programs by asserting properties of units whose bodies may not yet be available for analysis.

## 2.1 Additional Language Restrictions

Very few rules additional to those either that apply to sequential SPARK or imposed by the Ravenscar Profile are required for analysis purposes. Currently, the main additional restrictions are as follows:

- 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.
- 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).
- Protected elements must be initialized at declaration.
- Each Ada task must have a plain loop with no exits as its final statement.
- The attribute E'Count is not supported. (We believe that this restriction may also be removable.)

## 2.2 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 which 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 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 shown in figure 1.

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

| **Property** | **Applies to** | **Meaning** |
|---|---|---|
| Priority | own protected | the actual Ada priority that the object *will* have |
| Suspendable | own protected | a predefined suspension object or a protected object with an entry |
| Interrupt | own protected | object contains one or more interrupt handlers |
| Protects | own protected | identifies unprotected objects that are accessed only from the protected object, and inherit its protection |
| Suspends | procedure or task | the operation may suspend on the listed suspension objects and/or protected objects |
| Delay | procedure | the operation may execute a delay statement |

*Figure 1 – Definition of Properties*

## 3 Constructing Concurrent Programs

Existing static analysis techniques for sequential code can verify that the implementation of the compilable Ada part of a SPARK program conforms to its design, as expressed in its SPARK annotations. These techniques support data flow analysis, information flow analysis, and proof that includes the use of pre and post conditions and assertions. We have enhanced these techniques for Ravenscar programs that include tasks, protected objects and interrupt handlers, and the enhancements to the SPARK annotations are also designed to be extensible to accommodate future integration with schedulability analysis and model checking tools.

The enhancements to SPARK in this area can be categorised as:

1. thread-specific analysis, where the term “thread” may denote the Environment task, an Ada task, or an interrupt handler;
2. partition-wide analysis, where the term “partition” is used to denote an active Ada 95 partition.

## 3.1 Thread-specific Analysis

The data and information flow analysis techniques used in sequential SPARK are described in [[7]]. At the thread level of execution, where we are concerned with an individual task or subprogram, these techniques are 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, in the thread-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 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 by the same thread. For example, if we foolishly try to exchange the values of variables X and Y using protected object P as a temporary store:

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

then we find that instead of this 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 object, 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.

## 3.2 Partition-wide Analysis

The extension of the SPARK annotation system to support partition-wide analysis is extremely valuable. In sequential SPARK, the annotations provide the means to express the design of the program such that it is possible to perform analysis to show that the actual code meets this design. This facility is even more important in concurrent programs, since the flow of information between tasks and interrupt handlers through protected objects may be hard to verify without tool support.

The extension provides additional *global* and *derives* annotations that allow the programmer to describe the intended flow relation for the entire partition. The annotations express primarily the dependencies of all partition outputs to its inputs. This intended behaviour is compared by the tool with a flow relation that is constructed from the interactions between all threads of control (Ada tasks, the Environment task, and interrupt handlers) via the protected objects, to ensure that the actual interactions achieve the same effect. The construction of this flow relation is performed as follows:

1. 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.
2. 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 [[8]]). The ability to name the source of interrupts is especially useful if a protected type declares more than one handler, or if there are multiple objects of the same type.
3. 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.
4. 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 the value of D is detected.

It is important to note that the automatic construction of the partition flow relation does not require access to any Ada bodies - the information in the annotations present in the specifications is sufficient. We can therefore check that our program is properly constructed before getting involved in implementation details. The veracity of each annotation is checked when the bodies are written.

As well as providing a description of the overall intended behaviour, the partition-level annotation provides some protection from the nasty error of program incompleteness (see point 3 in section 4.2). If an active component of the program is omitted then its effects will not be included in the calculated partition flow relation, which is then unlikely to agree with the claimed relation in the partition annotation.

When this level of analysis is further enhanced by the use of response time analysis and model checking tools to analyse temporal behaviour, a comprehensive verification of the concurrent program can be realised.

## 4 Freedom from Run-Time Errors

The Ravenscar 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. This section briefly restates each problem and the approach that has been incorporated into SPARK for its elimination. A more comprehensive description of the SPARK approach can be found in [[5]].

### 4.1 Run-time Exceptions

**Unhandled Exceptions** An unhandled exception in a concurrent program may cause:

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

The existing definition of SPARK is sufficient to show proof of absence of run-time errors in sequential code due to language-defined exceptions (see [[6]]).

**Exceptions Due to Concurrency** There are four applicable concurrency checks defined either by Ada 95 or by the Ravenscar Profile that cause Program_Error exception to be raised if they fail.

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.

We identify all protected objects that may be called directly or indirectly from each task (including the Environment Task) and from each interrupt handler,. The priority of each protected object and task is required to be a static value. We can thus ensure that each protected object call chain does not cause a priority ceiling violation. Note that this approach has also been successfully applied in design methods such as HRT-HOOD [[9]].

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

3. The maximum number of calls that are queued concurrently on a protected entry is one.

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, without consideration of temporal aspects.

4. A potentially blocking operation shall not be executed by a protected action.

We detect calls to potentially-blocking operations from within a protected action in SPARK using the transitive *delay* or *suspends* property of a procedure.

### 4.2 Erroneous Behaviour

The following three categories of erroneous behaviour are addressed in SPARK:

1. Use of unprotected shared variables - if two tasks share an unprotected variable, the resulting program may be erroneous.

In SPARK, unprotected data *can never* be shared by virtue of allowing at most one task to access each unprotected global variable. We also prohibit protected objects from accessing unprotected state.

2. Race conditions during program elaboration

Section 6.2.2 of [[1]] provides a detailed explanation of the risk of race conditions during elaboration of concurrent programs. 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 library package body elaboration code. In SPARK, all *protected* variables must be statically initialized. In the case of *unprotected* variables, we ensure that the unprotected state is initialised only by a single Ada task, and not by library package elaboration code.

3. 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.

This kind of error is mitigated in SPARK by the partition flow relation that is discussed in section 3.2.

### 4.3 Implementation-Defined Behaviour – Task Termination

The Ravenscar Profile includes the Restrictions pragma *No_Task_Termination*, but the dynamic effect of task termination is implementation-defined. In SPARK, task termination is prevented by the rule requiring each task to end with a plain loop with no exit statements and by the static elimination of run-time exceptions from the program.

## 5 Short Example

The manner in which SPARK eliminates the runtime errors described above, and the additional analysis that can be achieved at partition level, are illustrated using a small example program. 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 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.

```
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;
```

The annotations in package `User` tell 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 three interrupt handling procedures with a programmer-selected name that will be use in the partition wide flow analysis. Finally, the priority of the object is announced.

```
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 “this instance”
      --# 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;
```

The annotations in package `Display` tell 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 included external objects such as those with associated address clauses or pragma Import, so the *protects* property may be used to announce protection for an otherwise unprotected own variable. Finally, we consider the main timing package:

```
package Timer
--# own protected Operate (suspendable);
--#     task TimingLoop : TT;
is
   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 runtime errors outlined in section 4. Some errors are only detected when the main program is analysed, but these checks require access only to the package specifications *withed* by the main program; at no stage is a complete, linkable closure of the entire program required.

Finally, we state the intended flow relation for the partition, using the supplied names from the *interrupt* property list in the annotation for `Buttons`:

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

This defines all three user buttons affecting the displayed time, and the `Reset` button *not* affecting `Timer.Operate` which controls whether the clock is running. This relation is compared to that computed from the annotations for each task and interrupt handler in the program, as described in section 3.2, and any discrepancy is reported as an error. For example, if the package `User` containing the interrupt handlers were inadvertently omitted from the main program context clause, this check would fail, even though the build of the Ada program would succeed.

## 6 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 analysable timing properties. The extension of SPARK to encompass the profile provides a way of statically showing such a program to be correctly constructed and free from run-time errors. This has been implemented with only a few minor additional restrictions to those in the profile. The combination of these techniques with suitable, certifiable, run-time support must represent the most rigorous environment for producing high-integrity concurrent programs.

## References

[1] Burns, Alan; Dobbing Brian; and Vardanega, Tullio: *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*.Praxis Critical Systems 2002.

[4] Barnes, John: *High Integrity Software - the SPARK Approach to Safety and Security* Addison Wesley Longman, ISBN 0-321-13616-0. 2003.

[5] Amey, Peter and Dobbing, Brian: *High Integrity Ravenscar*, 8th Ada-Europe International Conference on Reliable Software Technologies, Lecture Notes in Computer Science, vol. 2655, Springer-Verlag, 2003, Rosen, Jean-Pierre and Strohmeier, Alfred (Editors).

[6] Chapman, Rod; Amey, Peter: *Industrial Strength Exception Freedom*. Proceedings of ACM SIGAda 2002 (also downloadable from http://www.sparkada.com)

[7] Bergeretti and Carré: *Information-flow and data-flow analysis of while-programs* ACM Transactions on Programming Languages and Systems, pp37-61, 1985.

[8] Amey, Peter: *A Language for Systems not Just Software* Proceedings of ACM SIGAda 2001(also downloadable from http://www.sparkada.com)

[9] Burns, Alan; and Wellings, Andy: *HRT-HOOD: A Structured Design Method for Hard Real-Time Ada Systems*, Elsevier ISBN: 0-444-82164-3, 1995