Position paper: Completing the Ravenscar Profile

Stephen Michell
Maurya Software
29 Maurya Court Ottawa,
Ontario, Canada K1G 5S3
email steve@maurya.on.ca

Abstract

There are some interactions between the Ravenscar Tasking Profile and sequential Ada language constructs which can cause undesirable behaviour for high integrity systems. We examine some of the interactions and discuss potential changes to the definition of the Ravenscar profile to correct the deficiencies found.

1 Introduction

The Ravenscar Profile has generated a significant amount of interest in the hard real time, embedded, and high integrity (safety critical and security communities. It has been the subject of ongoing research [Fowler 1999] [LAM 1999] [LA 2000] and has at least two commercial implementations.

Current thinking in the Ada community is that the Ravenscar Profile should be made an ISO technical report or standard and incorporated into a revision of Ada in the 2005 time frame.

While Ravenscar has generated considerable interest and seems to be filling the role envisioned for it as the work of a specialised group, the situation changes significantly when such a profile is considered for inclusion as an integral part of the Ada language. In that context the profile must be cohesive in the context of the complete language and must address all of the relevant issues associated with tasking; issues such as memory management, shared variables, exceptions, task elaboration and activation, improved specification of bounded behaviours, indefinite and controlled types and scope closure/finalization.

Before Ravenscar is considered as a secondary standard to the Ada standard or is included in any update to ISO 8642:1995, these issues need to be addressed in the Ravenscar specification. This paper addresses the issues mentioned above and proposes how Ravenscar can address them.

Since the Ravenscar Tasking Profile is published [Burns 1999], we will not reiterate the specification here.

2 Memory Management

Most traditional long-lived, embedded, real-time and high integrity systems abhor dynamic memory. This presents a challenge since dynamic memory allocation and deallocation are an integral part of Ada. The Ravenscar-Tasking Profile [Burns 1999] assumed that programs and implementations would not use dynamic memory, but did not take explicit steps to enforce this in the profile specification. Indeed, some programs may wish to use Storage Pools, and at least one high integrity Ada implementation (AONIX) claims to have a storage pool capability for its high integrity product. There are also a number of Ada constructs which may use dynamic storage (eg - return of unconstrained types) in some implementations. Indeed, even the HRG has recognized that storage pools may be needed in high integrity systems [HRG 1998].

For concurrent systems the allocation and deallocation of memory must be atomic operations from the point of view of any combination of tasks that could be accessing the same storage pool or dynamic memory. For a Ravenscar runtime kernel to support such memory usage, either a new locking paradigm must be specified or the profile must embed all such accesses inside a protected object.

At the present time the Ravenscar Tasking Profile is silent on the matter of memory management. To address the issue, the Ravenscar Profile could chose one of the following alternatives.

1. Explicitly forbid memory management. This means that the Ravenscar Tasking Profile specification must also explicitly include the restrictions No_Implicit_Heap_Allocation, which prohibits actions such as return of unconstrained types where the implementation would use dynamic memory for transient

objects or other objects, and No_Allocators, which prohibits explicitly allocated dynamic memory.

2. Explicitly forbid use of the heap - i.e. No_implicit_Heap_Allocation and the use of "new" for types that do not have a storage pool, but permit allocation of objects from storage pools.

Each of these alternatives is analysed below.

2.1 Analysis

Item 1 above means that there is no interaction between a Ravenscar kernel scheduler and memory allocation and is the simplest to specify and implement. Unfortunately, this may be too restrictive for real programs, especially ones that intend to use storage pools.

Item 2 requires that the kernel specification enforce atomic memory allocation. In this case the Ravenscar profile could specify that all memory allocation have the semantics of a call to a protected object dedicated to a single storage pools, or could specify a unique protocol.

The use of protected operations to specify the semantics of memory operations from storage pools has scheduling implications. (While not strictly a signalling event as defined in ARM 9.10, the initiation and completion of a protected operation are scheduling points for a kernel since priority changes are required and task scheduling status must be re-evaluated.) It is for this reason that Ravenscar must specify whether or not storage pool support is included in the model and what model is provided to guarantee atomic access.

It is our belief that storage pools are a necessary Ada functionality, even for embedded and real time systems, and that Ravenscar must therefore specify how storage pool allocation and deallocation is handled by a compliant kernel.

Whatever choice is made however, the Ravenscar Profile must specify the semantics of the memory model that it chooses.

3 Shared Variables

Shared variables are a fundamental intertask communication mechanism in Ada. Ada provides the pragmas "volatile", "atomic" and "atomic_components" to guarantee that such objects are correctly updated. Ravenscar explicitly permits the pragmas Volatile and Atomic, but does not address how a kernel might support these when the variables being managed are more complex that a single processor word. (Note that implementations must generate compile time errors when these pragmas are not supported by the implementation for the selected object size. The relevant issue is that the hardware support for atomic update may differ from the implementations, leaving the kernel to implement locking semantics for objects which fall into this class.)

In order to implement these pragmas, a compiler code generator must have some knowledge of the underlying architecture and possibly how the runtime is mapped onto the processor (eg - use of hardware cache, processor modes, or if memory is shared between multiple processors). It is often the case that pragma atomic cannot be implemented without task scheduling support from the runtime.

For example, a packed array of booleans with pragma atomic_components applied will require several machine instructions to compare or update any subset of the variable. The risk of interleaving operations on the object from different tasks demands an kernel lock be provided and used.

To date there have been two attempts to formally verify the Ravenscar tasking profile[Fowler 1999] [LA 2000], and neither of them has considered the scheduling implications of shared variables for Ravenscar. This clearly demonstrates that there is a need for the Ravenscar Tasking Profile to explicitly address the issue of shared variables in a Ravenscar-compliant system.

For a Ravenscar kernel, the only locking mechanism expected to be provided is the protected object. This means that locking mechanisms for shared objects must be expressed in terms of protected objects. Therefore we propose that the Ravenscar Tasking Profile state that all shared variables to which pragma atomic or atomic_components applies where multiple processor instructions are required to update or compare the object or its components be managed by the kernel as though protected by a protected object with ceiling priority priority'last with a protected procedure for each operation (such as assignment, equality, less than, etc.)

4 Exceptions

Even though Ravenscar attempted to define the tasking profile to not require exceptions, it must be noted that exceptions are a language feature of Ada that cannot be avoided. Many exceptions commence with hardware exceptions or traps that are delivered to the executing task and must be handled. The Ada semantics for exception propagation and handling specify how such exceptions are handled once acquired by the software.

There are essentially two ways that exceptions interact with concurrency in Ada; within the conext of a single task, and in contexts between tasks or between the kernel and a task.

Most exceptions are raised, propogated and handled in the context of the operation of a single task (such as Constraint_Error, Program_Error or explicit exceptions). Except where such exceptions interact with a protected operation, the elaboration of a task or the termination of a task, there are no scheduling issues.

Exceptions raised by the runtime environment or by a task but affecting other tasks have a deleterious effect on the predictability of a program and must be identified and eliminated. Situations that can result in such exceptions are as follows.

1. When a task is queued on an entry and another task calls the same entry before the first one is released, a bounded error results as defined by the Ravenscar Profile [Burns 1999]. The bounded behaviour could be the detection of the situation at compile time and a compile-time error, creation of an exception in the second calling task, the loss of knowledge of the state of the first queued task, or an exception in the first task.

2. Exceptions raised during elaboration of library-level tasks (caused perhaps by discriminant evaluation of a task being elaborated) will cause the main task performing the elaboration to raise an exception. Tasks elaborated and activated in declarative regions elaborated before the one under consideration will be active.

3. Ada95 permits interrupts to be dynamically attached, detached and replaced. These attach and detach operations can cause exceptions to be raised in the task executing these calls, due to events or sequences events that are not correctable by the task that caught the exception.

4.1 Analysis

The general issue about the interaction of exceptions with protected operations, including entry guard evaluation and exception propogation, was addressed by Lundqvist, Asplund and the author [LAM 1999]. In this paper, our formal model of protected objects included the complete semantics of exceptions raised in a protected operation and propagated to the caller. We believe that this specification is sufficient for the specification of exceptions and protected objects.It is the case, however, that exceptions may also be propagated to the main body of a task and cause the task to terminate. Attempts to terminate a task in Ravenscar is a bounded error. It is not permissible to raise an exception in other tasks (including the mainline) however, so we believe that Ravenscar should specify the bounds of this error.

The first identified issue, attempts to queue a task on an entry queue that is full, is presently a bounded error in Ravenscar. Possible bounds of such behaviour is the loss of the knowledge that one of the waiting tasks is queued for a protected entry, conceivably resulting in the perpetual blockage of the "lost" task. The current permissiveness of making failure to obey this restriction only a bounded error can be improved. Evaluation of queue entry guards and the state of the queue is a protected operation, hence the runtime always knows when Ravenscar restrictions have been exceeded and can faithfully raise Program_Error.

The original definition of Ravenscar left this as a bounded error in the belief that some programs using Ravenscar may not wish to handle exceptions. We argue, however, that this puts an obligation on the application to show that situations which would result in an exception are avioded; it does not place a burden on the runtime to avoid raising exceptions where they are warranted.

Therefore we believe that this situation must result in an exception being raised and propose that Ravenscar make it the stated behaviour when a task attempts to queue to an entry that already has a waiting task.

The second issue, exceptions during task elaboration, cause exceptions to be propogated to the elaborator of the task. In Ravenscar, this can only happen during startup before the mainline commences execution. Even without exceptions being raised, there may be race conditions between elaborating tasks that. It is our belief that the Ravenscar Tasking Profile should provide more explicit elaboration control to ensure that explicit control of task elaboration and startup.

The third issue, the dynamic attachment and detachment of exception handlers, has serious implications to programs. The state of interrupt management hardware could cause premature interrupts as the handler is being attached. Detachment of a handler could occur in an order that does not correctly match the order of attachment, resulting in the exception Program_Error being raised. There is no known algorithm to ensure that such attachments and detachments do not raise exceptions in the general case.

For these reasons, we believe that Ravenscar should either explicitly forbid interrupt attach and detach handlers, leaving only the static attachment of such handlers; or should limit the use of dynamic handler attachment to a single event, thereby severly limiting the ways that such exceptions can be raised.

In general, Ravenscar should avoid bounded errors. If the application can detect an erroneous situation, Ravenscar should insist that an exception be raised. The raising and handling of exceptions is not an issue for Ravenscar; they need to be raised, propogated and handled as a normal part of the language, relying upon applications that need to eliminate exceptions to use analytical techniques for this elimination.

5 Scope Management

The termination of a scope (subprogram or block) can be a complex activity in Ada. At a minimum the following activities may be performed at each scope closure:

- exception traps, exception evaluation, and exception propagation,

- wait for any locally declared tasks to terminate

- finalisation of objects whose existence is terminated within the scope.

The first item, exception processing, is discussed in section 4 above. There are no new requirements that arise because of the interaction of exceptions and scope closure.

The second item, the interaction of tasks and scope closure, is precluded in Ravenscar because nested task objects and allocators are forbidden. Similarly, the interaction between scope and protected objects is irrelevant since nested protected objects could interact with at most a single task (the one executing the scope under consideration) and are hence uninteresting.

The third item, finalisation of objects, is a significant area that needs consideration. In general finalisation is coupled with garbage collection. Depending upon the implementation, it may not occur on the thread of the task closing the scope if the work to be done is significant. Situations where this can occur are associated with nested task objects, nested protected objects, and nested objects derived from controlled types. Nested tasks and protected objects were discussed above, hence we are left to consider nested objects of types derived from controlled types.

This discussion leads us to conclude that Ravenscar should prohibit the declaration of objects of types derived from Ada.Finalization.Controlled in scopes that are deeper than library level, and prohibit allocators to such types. The effect of banning the declaration of derivatives from controlled types from all such places makes controlled types uninteresting because they can never go out of scope. It therefore seems reasonable to prohibit controlled types in Ravenscar compliant runtime systems.

6 Conclusions

This paper has addressed the outstanding situations in Ada where language features, though not strictly concurrent themselves, interact with concurrency in ways that may lead to unsafe situations. Before Ravenscar can become fully integrated into an Ada language definition, they must be considered and appropriate action taken to address them in the context of Ravenscar. Each issue either needs to be explicitly addressed and kernel behaviour specified, or additional restrictions are required to make sure that the events are not possible.

Reference

[ARM 1995] ISO/IEC/JTC1 8652:1995 Ada Programming Language Reference Manual, ISO 1995

[Burns 1999] Burns A. The Ravenscar Profile. Ada Letters, December 1999

[Fowler 1999] Fowler S. A Development Method for Trusted Real Time Kernels. PhD Dissertation, York University, York UK. 1999

[Kamrad 1999] Kamrad M. An Ada Runtime System Implementation of the Ravenscar Profile for a High Speed Application Layer Data Switch. Reliable Software Technologies Proceedings of the Ada Europe Conference, Springer Vorlag, 1999

[HRG 1998] Guidance on the Use of the Ada Programming Language in High Integrity Systems. ISO/IEC/JTC1 Technical Report 15942, ISO 1998

[LA 2000] Lundqvist K., Asplund L. A Ravenscar-Compliant Run Time for Safety Critical Systems. Real Time Systems Journal, Stockholm, Sweden, Kluwer Acedemic Publishers Group, 2000

[LAM 1999] Lundqvist K., Asplund L., Michell S. A Formal Model of the Ravenscar Tasking Profile: Protected Objects. Reliable Software Technologies Proceedings of the Ada Europe Conference, Springer Vorlag, 1999