Technical Specification

ISO/IEC TS 24718

First edition
2025-01

Information technology — Programming languages — Guidance for the use of the Ada Ravenscar Profile in high integrity systems

Technologies de l'information — Langages de programmation — Guide pour l'usage du profil "Ada Ravenscar" dans les systèmes de haute intégrité

Reference number
ISO/IEC TS 24718:2025(en)

ISO copyright office
CP 401 • Ch. de Blandonnet 8
CH-1214 Vernier, Geneva
Phone: +41 22 749 01 11
Email: copyright@iso.org
Website: www.iso.org

Published in Switzerland

Contents

Page

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work.

The procedures used to develop this document and those intended for its further maintenance are described in the ISO/IEC Directives, Part 1. In particular, the different approval criteria needed for the different types of document should be noted. This document was drafted in accordance with the editorial rules of the ISO/IEC Directives, Part 2 (see www.iso.org/directives or www.iec.ch/members_experts/refdocs).

ISO and IEC draw attention to the possibility that the implementation of this document may involve the use of (a) patent(s). ISO and IEC take no position concerning the evidence, validity or applicability of any claimed patent rights in respect thereof. As of the date of publication of this document, ISO and IEC had not received notice of (a) patent(s) which may be required to implement this document. However, implementers are cautioned that this may not represent the latest information, which may be obtained from the patent database available at www.iso.org/patents and https://patents.iec.ch. ISO and IEC shall not be held responsible for identifying any or all such patent rights.

Any trade name used in this document is information given for the convenience of users and does not constitute an endorsement.

For an explanation of the voluntary nature of standards, the meaning of ISO specific terms and expressions related to conformity assessment, as well as information about ISO's adherence to the World Trade Organization (WTO) principles in the Technical Barriers to Trade (TBT) see www.iso.org/iso/foreword.html. In the IEC, see www.iec.ch/understanding-standards.

This document was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 22, *Programming languages, their environments and system software interfaces*.

This first edition cancels and replaces the first edition (ISO/IEC TR 24718:2005).

The main changes are as follows:

— a relatively minor change to the use of a newer syntactic form for specifying aspects of entities, such as the relative priority of a task, rather than the prior use of pragmas;

— a more important change resulting from updates to the definition of the Ravenscar profile, in which support for multiple cores is now included. The primary change is to specify that all assignments of tasks to CPUs are static. In addition, some language-defined facilities are specified as not required or included in the profile for the sake of ensuring a relatively simple run-time library implementation.

Any feedback or questions on this document should be directed to the user's national standards body. A complete listing of these bodies can be found at www.iso.org/members.html and www.iec.ch/national-committees.

Introduction

0.1 General

There is an increasing recognition that the software components of critical real-time applications can be demonstrated as predictable. This is particularly the case for a hard real-time system, in which the failure of a component of the system to meet its timing deadline can result in an unacceptable degradation of the whole system. The choice of a suitable design and development method, in conjunction with supporting tools that enable the real-time performance of a system to be analysed and simulated, can lead to a high level of confidence that the final system meets its real-time constraints.

Traditional methods used for the design and development of complex applications, which concentrate primarily on functionality, are increasingly inadequate for hard real-time systems. This is because non-functional requirements such as dependability (e.g. safety and reliability), timeliness, memory usage and dynamic change management are left until too late in the development cycle.

The traditional approach to formal verification and certification of critical real-time systems has been to dispense entirely with separate processes, each with their own independent thread of control, and to use a cyclic executive that calls a series of procedures in a fully deterministic manner. Such a system becomes easy to analyse but is difficult to design for systems of more than moderate complexity, inflexible to change, and not well suited to applications where sporadic activity can occur and where error recovery is important. Moreover, it can lead to poor software engineering if small procedures must be artificially constructed to fit the cyclic schedule.

The use of the Ada programming language has proven to be of great value within high-integrity and real-time applications, albeit via language subsets of deterministic constructs, to ensure full analysability of the code. Such subsets have been defined for ISO/IEC 8652:1987 (conventionally known as “Ada 83” by language users), but these have excluded tasking on the grounds of its non-determinism and inefficiency. Subsequent advances in the area of schedulability analysis have allowed hard deadlines to be checked, even in the presence of a run-time system that enforces pre-emptive task scheduling based on multiple priorities. This valuable research work has been mapped onto a number of new Ada constructs and rules that have been incorporated into the Real-Time Annex of the Ada language Standard (ISO/IEC 8652:2023, Annex D). This evolution has opened the way for these tasking constructs to be used in high integrity subsets while retaining the core elements of predictability and reliability.

The Ravenscar profile is a subset of the tasking model as defined in ISO/IEC 8652:2023. It is restricted to meet the real-time community requirements for determinism, schedulability analysis and memory-boundedness, and is also suitable for mapping to a small and efficient run-time system that supports task synchronization and communication, and which can be certifiable to the highest integrity levels. The concurrency model promoted by the Ravenscar profile is consistent with the use of tools that allow the static properties of programs to be verified. Applicable verification techniques include information flow analysis, schedulability analysis, execution-order analysis and model checking. These techniques allow analyses of a system to be performed throughout its development life cycle, thus avoiding the common problem of discovering only during system integration and testing that the design fails to meet its non-functional requirements.

It is important to note that the Ravenscar profile is silent on the non-tasking (i.e. sequential) aspects of the language. For example, it does not dictate how exceptions should, or should not, be used. For any application in the intended domain, it is likely that constraints on the sequential part of the language will be required. These can be due to other forms of static analysis to be applied to the code, or to enable worst-case execution time information to be derived for the sequential code. See ISO/IEC TR 15942 for a detailed discussion on all aspects of static analysis of sequential Ada.

The Ravenscar profile has been designed such that the restricted form of tasking that it defines can be used even for software that should be verified to the very highest integrity levels. The Ravenscar profile has already been included in ISO/IEC TR 15942.

0.2 Structure

The document is organized as follows. The motivation for the development of the Ravenscar profile is given in Clause 4. Clause 4 also includes the definition of the profile as specified by ISO/IEC 8652:2023 (the Ada

Standard); the definition is included here for convenience, but this document is not the definitive statement of the profile. In Clause 6, the rationale for each aspect of the profile is described. Examples of usage are then provided in Clause 7. The need for verification is an important design goal for the Ravenscar profile: Clause 8 reviews the verification approach appropriate to Ravenscar programs. Finally, in Clause 9 an extended example is given.

0.3 Conventions

For all Ada-related terms, this document follows the style of the ISO/IEC 8652:2023 (the Ada standard): it uses a distinct font where there is a reference to defined syntax entities (e.g. `delay_relative_statement`).

Information technology — Programming languages — Guidance for the use of the Ada Ravenscar Profile in high integrity systems

1 Scope

This document provides guidance on the use of the Ravenscar profile for concurrent Ada software intended for verification up to, and including, the very highest levels of integrity.

To this end, this document provides a complete description of the motivations behind the Ravenscar profile, to show how conformant programs can be analysed, and to give examples of usage.

This document is aimed at a broad audience, including application programmers, implementers of run-time systems, those responsible for defining company or project guidelines, and academics. Familiarity with the Ada language is assumed.

2 Normative references

There are no normative references in this document.

3 Terms and definitions

For the purposes of this document, the following terms and definitions apply.

ISO and IEC maintain terminology databases for use in standardization at the following addresses:

— ISO Online browsing platform: available at https://www.iso.org/obp

— IEC Electropedia: available at https://www.electropedia.org/

3.1
atomic
type of operation performed by a task that is guaranteed to always produce the same effect as if it were executed in total isolation and without interruption

3.2
blocked
waiting for mutually exclusive access to a shared resource that is currently held by a lower priority task

3.3
bounded error
implementation- or language-defined error in the application program whose effect is predictable and documented

3.4
ceiling priority
static default priority of a shared resource greater than or equal to the highest priority of any accessing task

3.5
context switch
replacement of one task by another as the executing task on a processor

3.6
critical region
sequence of statements that appear to be executed indivisibly

3.7
critical task
task whose *deadline* (3.10) is significant and whose failure to meet its deadline can cause system failure

3.8
cyclic executive
system scheduler that uses procedure calls to execute each periodic process in a predetermined sequence at a predetermined rate

3.9
cyclic task
periodic task
task whose execution is repeated based on a fixed period of time

3.10
deadline
maximum time allowed to a task to produce a response following its invocation

3.11
deadlock
situation in which a group of tasks (possibly the whole system) block each other permanently

3.12
dynamic testing
analysis method that determines properties of the software by observing its execution

Note 1 to entry: See *static analysis* (3.28).

3.13
epilogue
code executed by the Ada run-time system to finish service to protected calls

3.14
event-triggered task
task whose invocation is triggered either by an asynchronous action by another task, or by an external stimulus such as an interrupt

3.15
finalization
operation which occurs for controlled objects at the point of their destruction

Note 1 to entry: See ISO/IEC 8652:2023, 7.6 for further information.

3.16
jitter
variation in time between the occurrence of a periodic event and a period of the same frequency

3.17
livelock
situation in which several tasks (possibly comprising the whole system) remain ready to run, and execute, but fail to make progress

3.18
liveness
system property implying that a set of tasks will reach all desirable states

3.19
mode change
change in operating characteristics of a system that requires a co-ordinated change in the operation of several different processes in the system

3.20
monitor
module containing one or more *critical regions* (3.6) such that all variables potentially accessed under mutual exclusion are hidden and all procedure calls are guaranteed to execute with mutual exclusion

3.21
mutex
locking mechanism used to ensure mutually exclusive access to a shared resource

3.22
overhead
execution time within the Ada run-time system which is included in the schedulability analysis

3.23
priority inversion
situation in which a high-priority task is *blocked* (3.2) waiting for a shared resource (including the processor itself) currently in use by a low-priority task

3.24
race condition
timing condition that causes processes to operate in an unpredictable sequence so that operation of the system can be incorrect

3.25
ready
state of a task when it is no longer *suspended* (3.29) (but not necessarily executing, depending on available processor resources)

3.26
safety
system property implying that a set of tasks cannot reach any undesirable state from any desirable state

3.27
sporadic task
event-triggered task (3.14) with defined minimum inter-arrival time

3.28
static analysis
group of analysis techniques that determine properties of the system from analysis of the program source code

Note 1 to entry: See *dynamic testing* (3.12).

3.29
suspended
state of a task when its execution is stopped due to execution of a language-defined construct that waits for a given time (e.g. a delay statement) or an event

3.30
suspending operation
operation which causes the current task to be *suspended* (3.29) until released by another task, a timer event or an interrupt handler

3.31
suspension object
Ada construct which is used for basic task synchronization, i.e. suspend and resume, which does not involve data transfer

Note 1 to entry: See ISO/IEC 8652:2023, D.10.

3.32
time-triggered task
task whose invocation is triggered by the expiry of a delay set by that task

3.33
worst-case execution time
maximum bound on the time required to execute some sequential code

4 Motivation for the Ravenscar profile

4.1 General

Before describing the Ravenscar profile in detail, this clause explains some of the reasoning behind its features. These primarily come from the need to be able to verify concurrent real-time programs, and to have these programs implemented reliably and efficiently.

This clause examines mainly scheduling theory, as this is the main driver for the definition of the restrictions of the Ravenscar profile. In addition, there is a subclause that summarizes other program verification techniques that can be used with the profile.

4.2 Scheduling theory

4.2.1 General

State-of-the-art research in scheduling theory has found that accurate analysis of real-time behaviour is possible given a careful choice of scheduling or dispatching method together with suitable restrictions on the interactions allowed between tasks. An example of a scheduling method is fixed priority pre-emptive scheduling, in which each process has a static priority and the scheduler ensures that the currently selected process is the ready process with the highest priority. Examples of analysis schemes are rate monotonic analysis (RMA)[5] and response time analysis (RTA).[6]

Fixed-priority pre-emptive scheduling is normally used with a Priority Ceiling Protocol (PCP) to avoid unbounded priority inversion and the risk of circular deadlock on access to shared resources. Adoption of the PCP provides a model suitable for the analysis of concurrent real-time systems. The approach supports cyclic and sporadic activities, the notions of hard, soft, firm, and non-critical application components, and controlled communication and synchronization among application components. It is also scalable to programs for distributed and multiprocessor systems.

Tool support exists for RMA and RTA, and for the static simulation of concurrent real-time programs. The primary aim of analysing the real-time behaviour of a system is to determine whether it can be scheduled in such a way that it is guaranteed to meet its timing constraints. Whether the timing constraints are appropriate for meeting the requirements of the application is not an issue for scheduling analysis. Such verification requires a more formal model of the program and the application of techniques such as model checking, see 4.5.

4.2.2 Tasks characteristics

The various tasks in an application will each have timing constraints, which correspond to deadlines.

Each task is classified into one of the following four basic levels of criticality according to the importance of meeting its deadline.

— Hard: a hard deadline task is one that is required to meet its deadline. The failure of such a task to meet its deadline can result in an unacceptable failure at the system level.

— Firm: a firm deadline task is one that is required to meet its deadline under “average” or “normal” conditions. An occasional missed deadline can be tolerated without causing system failure (but can

result in degraded system performance). There is no value, and thus there is a system-level degradation of service, in completing a firm task after its deadline.

— Soft: a soft deadline task is also one that is required to meet its deadline under “average” or “normal” conditions. An occasional missed deadline can be tolerated without causing system failure (but can result in degraded system performance). There is value in completing a soft task even if it has missed its deadline.

— Non-critical: a non-critical task has no strict deadline. Such a task is typically a background task that performs activities such as system logging. Failure of a non-critical task does not endanger the performance of the system.

4.2.3 Scheduling model

At any moment in time, some tasks can be ready to run, meaning that they are able to execute instructions if processor time is made available. Others are suspended, meaning that they cannot execute until some event occurs, or blocked, meaning that they await access to a shared resource that is currently exclusively owned by another task. Suspended tasks can become ready synchronously (as a result of an action taken by a currently running task) or asynchronously (as a result of an external event, such as an interrupt or timeout, that is not directly stimulated by the current task).

With priority-based pre-emptive scheduling on a mono-processor, a priority is assigned to each task and the scheduler ensures that the highest priority ready task is always executing. If a task with a priority higher than the currently running task becomes ready, the scheduler performs a *context* switch, as soon as it can, to enable the higher-priority task to begin or resume execution. The term “pre-emptive” indicates that this can occur because of an asynchronous event (i.e. one that is not caused by the running task).

Tasks will normally be required to interact as a result of contention for shared resources, exchange of data, and the need to synchronize their activities. Uncontrolled use of such interactions can lead to a number of problems:

— Unbounded priority inversion (also known as blocking): where a high-priority task is blocked awaiting a resource in use by a low-priority task. As a result, ready tasks of intermediate priority can hold up the high priority task for an unbounded amount of time since they will run in preference to the low priority task that has locked the resource.

— Deadlock: where a group of tasks (possibly the whole system) block each other permanently due to circularities in the ownership of and the contention for shared resources.

— Livelock: where several tasks (possibly the whole system) remain ready to run, and do indeed execute, but fail to make progress due to circular data dependencies between the tasks that can never be broken.

— Missed deadline: where a task fails to complete its response before its deadline has expired due to factors such as system overload, excessive pre-emption, excessive blocking, deadlocks, livelocks or overruns.

The restricted scheduling model that is defined by the Ravenscar profile is designed to minimize the upper bound on blocking time, to prevent deadlocks and (via tool support) to verify that there is sufficient processing power available to ensure that all critical tasks meet their deadlines.

In this model, tasks do not interact directly, but instead interact via shared resources known as protected objects. Each protected object typically provides either a resource access control function (including a repository for the private data to manage and implement the resource), or a synchronization function, or a combination of both.

A protected object that is used for resource access control requires a mutual exclusion facility, commonly known as a monitor or critical region, where a maximum of one task at a time can have access to the object. During the period that a task has access to the object, the task is not allowed to perform any operation that can result in it becoming suspended. Ada directly supports protected objects and disallows internal suspension within these objects.

A protected object that is used for synchronization provides a signalling facility, whereby tasks can signal and/or wait on events. In the Ravenscar profile definition, the use of protected objects for synchronization by the critical tasks is constrained so that at most one task can wait on each protected object. A simplified version of wait/signal is also provided in the Ravenscar profile via the Ada real-time annex functionality known as suspension objects (see ISO/IEC 8652:2023, D.10). These can be used in preference to the protected object approach for simple resumption of a suspended task, whereas the protected object approach should be used when more complex resumption semantics are required, for example, including deterministic (race-condition-free) exchange of data between signaller and waiter tasks.

The Ravenscar profile definition ensures the absence of deadlocks by requiring use of an appropriate locking policy. This policy requires a ceiling priority to be assigned to each protected object that is no lower than the highest priority of all its calling tasks, and results in the raising of the priority of the task that is using the protected object to this ceiling priority value. In addition to the absence of deadlocks, this policy also allows an almost optimal time bound on the worst-case blocking time to be computed for use within the schedulability analysis, thereby eliminating the unbounded priority inversion problem. This time bound is calculated as the maximum time that the object is in use by lower-priority tasks. Therefore, the smaller the worst-case time bound for this blocking period, the greater the likelihood that the task set will be schedulable.

The use of priority-based pre-emptive dispatching defines a mechanism for scheduling. The scheduling policy is defined by the mapping of tasks to priority values. Many different schemes exist for different temporal characteristics of the tasks and other factors such as criticality. What most of these schemes require is an adequate range of distinct priority values. Ada and the Ravenscar profile ensure this.

The Ada programming language also provides another facility to help control object sharing: the atomic aspect. All reads and updates applied to an object marked with the atomic aspect are indivisible. Moreover, all tasks of the program (on all processors) that read or update an object marked atomic will see the same order of updates, as that marking also makes the object volatile. The language defines such reads and updates as interactions of the program with the external environment (memory in this case). However, for safe sharing of atomic objects, static assurance of a proper read/write protocol is highly recommended. In order to map Ada to the scheduling model being discussed here, however, protected objects are the primary and preferable abstraction as they are inherently safe.

4.3 Mapping Ada to the scheduling model

The analysis of an Ada application that makes unrestricted use of Ada run-time features including tasking rendezvous, select statements and abort is not currently feasible. In addition, the non-deterministic and potentially unbounded behaviour of many tasking and other run-time calls can make it impossible to provide the upper bounds on execution time that are used to perform schedulability analysis and scheduling simulation. Thus, Ada coding style rules and subset restrictions should be followed to ensure that all code within critical tasks is statically time-bounded, and that the execution of the tasks can be defined in terms of response times, deadlines, cycle times, and blocking times due to contention for shared resources.

The application is decomposed into a number of separate tasks, each with a single thread of control, with all interaction between these tasks identified. Each task has a single primary invocation event. The tasks are categorized as time-triggered (meaning that they execute in response to a time event), or event-triggered (meaning that they execute in response to a stimulus or event external to the task). If a time-triggered task receives a regular invocation time event with a statically-assigned rate, the task is termed periodic or cyclic.

Protected objects are introduced to provide mutually exclusive access to shared resources (e.g. for concurrent access to writable global data) and to implement task synchronization (e.g. via some event signalling mechanism). This decomposition is normally the result of applying a design methodology suitable to describe real-time systems.

In order to be suitable for schedulability analysis, the task set to be analysed should be static in composition with all its dependencies between tasks set via protected objects. Tasks nested inside other Ada structures incur unwanted visibility dependencies and termination dependencies. Therefore, this model only permits tasks to be created at the library level, at system initialization time.

Hence, in the Ravenscar profile, all tasks in the program are created at the library level.

Another consequence of requiring a static task set for schedulability analysis purposes is that the Ravenscar profile prohibits the dynamic creation of tasks and protected objects via allocators. This implies that the memory requirements for the execution of the task set (e.g. the task stacks) are resolved prior to, or during, the elaboration of the program. In addition, the Ravenscar profile prohibits the implementation from implicitly acquiring dynamic memory from the standard storage pool (see ISO/IEC 8652:2023, 13.11. The data structures that are required by the run-time system should either be declared globally, so that the memory requirements can be determined at link time, or in such a way as to cause the storage to be allocated on the stack (of the environment task) during elaboration of the run-time system.

The Ravenscar profile places no restrictions on the declaration of large or dynamic-sized Ada objects in the application other than prohibiting the implementation from implicitly using the standard storage pool to acquire the storage for these objects. The storage that holds such objects can be allocated on the task stack.

4.4 Non-preemptive scheduling and Ravenscar

The definition of the Ravenscar profile requires a pre-emptive scheduling of tasks. However, a similar profile can be defined that specifies non-pre-emptive execution. Much of the material and guidelines contained in this document would also apply to the non-pre-emptive case. Non-pre-emptive implementation for a mono-processor is in between the cyclic executive approach and the pre-emptive tasking approach with regard to ease of timing analysis, flexibility with regard to change, and responsiveness to asynchronous events. In common with the cyclic executive approach, there is no contention for shared resources, and there is no need to analyse the impact from asynchronous events. There is still, however, the need to break up long code sequences using voluntary suspension points (e.g. a `delay_until_statement` with a wakeup time argument that denotes a time in the past) to obtain reasonable responsiveness to asynchronous events.

4.5 Other program verification techniques

4.5.1 General

In addition to the provision of support for schedulability analysis, the rationale behind the Ravenscar profile definition is also to support other static program verification techniques, and to simplify the formal certification process. The following clause discusses these other techniques briefly.

4.5.2 Static analysis

Static analysis is recognized as a valuable mechanism for verifying software. For example, it is mandated for safety critical applications that are certified to the UK Defence Standard 00-55.[4] Industrial experience shows that the use of static analysis during development eliminates classes of errors that can be hard to find during testing. Moreover, these errors can be eliminated by the developer before the code has been compiled or entered into the project's configuration management system, saving the cost of repeated code review and testing which ensues from faults that are discovered during testing.

Static analysis as a technology has a fundamental advantage over dynamic testing. If a program property is shown to hold using static analysis, then the property is guaranteed for all scenarios. Testing, on the other hand, can demonstrate the presence of an error, but the correct execution of a test only indicates that the program behaves correctly for the specific set of inputs provided by the test, and within the specific context that the test harness sets up. For all but the simplest systems, exhaustive testing of all possible combinations of input values and program contexts is infeasible. Typically, test cases are devised to represent broad classes of inputs, so that tests can be created that use a representative value from each possible input class. However, complex program state contexts are usually only creatable during integration and system testing, when it can be very difficult to simulate all possible operational states. Further, the impact of correcting errors that are found only at this stage of the lifecycle is generally large in comparison to errors found during development.

There are many methods of static analysis. By using combinations of these methods, a variety of properties can be guaranteed for a program. The following list of forms of analysis is drawn from a study of a variety of

standards presented in ISO/IEC TR 15942. 8.3.1 discusses how these analyses can be applied in the context of a concurrent Ravenscar profile program.

— Control flow

  Control flow analysis ensures that code is well structured and does not contain any syntactically or semantically unreachable code.

— Data flow

  Data flow analysis ensures that there is no executable path through the program that would result in access to a variable that does not have a defined value. Data flow analysis is only feasible on code that has valid control flow properties.

— Information flow

  Information flow analysis is concerned with the dependencies between inputs and outputs within the code. It checks the specified dependencies against the implemented dependencies to ensure consistency. To be effective, information flow analysis should be performed with knowledge of the system requirements. It can be a powerful tool for demonstrating properties such as non-interference between critical and non-critical data.

— Symbolic execution

  Symbolic execution generates a model of the function of the software in terms of parallel assignments of expressions to outputs for each possible path through the code. This can be used to verify the code without the need for a formal specification.

— Formal code verification

  Formal code verification is the process of proving the code is correct against a formal specification of its requirements. Each operation is specified in terms of the pre-conditions that should be satisfied for the operation to be callable, and the post-conditions that hold following a successful call to the operation. The verification process demonstrates that, given the pre-conditions, execution of the operation always gives rise to the post-conditions. The level of proof depends on the information provided in the formal specification. This can vary depending on the aspects of the code that should be verified; this can vary from the proof of a single invariant right up to full functional behaviour.

  Proof of absence of run-time errors is a special form of formal code verification. This does not require the provision of a formal specification of the program. Instead, formal code verification techniques are used to demonstrate that, at every point in the code where a run-time error can occur, the pre-conditions on execution of that code and the current set of data values in the expression guarantee that the run-time error cannot occur. This is a very valuable property to be able to demonstrate, especially in systems where the occurrence of an unexpected run-time exception is generally unrecoverable, and the overhead of dynamic defensive mechanisms for preventing all such faults is unacceptable.

4.5.3 Formal analysis

The formal analysis of concurrent programs has been a fruitful research topic for a number of years. Current standard techniques allow many important properties of programs to be statically checked.

Concurrent programs, while more expressive than their sequential counterparts, have a number of distinct error conditions to address during program development. The most common of these is deadlock, where all processes are blocked on a synchronization primitive with no processes left to undertake the necessary unblocking actions. In general, a concurrent program should possess two important properties:

1. Safety: the system of tasks should not get into an unsafe (undesirable) state (for example, deadlock or livelock).

2. Liveness: all desirable states of the task are reached eventually (that is, useful progress should always be made).

In a real-time concurrent system, "liveness" becomes "bounded liveness" as desirable states should be reached by known deadlines.

Ada, like all other engineering languages, does not have its semantics defined in a formal mathematical way. Hence, it is recommended to link a model of the program with the program itself. This link cannot be formal but can be precise. The use of standard patterns for Ada tasks helps this linkage. The formal model can be derived from the code or, more likely in an engineering process, the model is derived from requirements, and the code is obtained via a series of refinements from the model.

There are two general forms for these models and two methods of extracting properties (behaviours) from these descriptions. First, an algebraic form can be used in one of the concurrency languages that does have formally defined semantics, such as Communicating Sequential Processes (CSP)[15] and Calculus of Communicating Systems (CCS).[16] The other more common approach is to view the program as a collection of state-transition systems.

Verification comes either from a proof theoretic approach or via model checking. An algebraic description can be proved to be deadlock-free, for example, by the use of a theorem prover. Alternatively, a state-transition description (or an algebraic one) can be exercised by an exhaustive search of the set of states the program can enter. This 'checking of the model' can deduce that all safe states, and no unsafe states, can be reached.

The disadvantage of model checking is that an explosion of states can make it impossible to terminate the search. However, there have been considerable (and continuing) advances in the tools for model checking, and now sizeable systems can be verified in a respectably small number of hours of processing time. Theorem proving does not have this problem, but it is a more skilled activity and theorem proving tools are not simple to use (i.e. the verification process is not automatic). A proof theoretic approach also has the advantage that it can show that a property is true "for any number of tasks", whereas model checking cannot generalize in this way. For example, it can show that it is true for six client tasks, say, but this claim does not hold automatically for seven or more tasks. Combinations of proof and model checking are possible and are the subject of current research.

For real-time systems, it is possible to add time to the concurrency model and to then validate temporal aspects of program. Timed versions of formalisms such as CSP exist and state-transition systems with clocks allow timing requirements to be expressed and subsequently verified by model checking. A common formalism for this type of state-transition system is called timed automata. Again, tool support for model checking sets of timed automata is well advanced. One of the very useful features of model checking tools is that they all produce a well-defined counter example for any failed check.

4.5.4 Formal certification

In order to achieve formal certification of a software architecture and of its Ada implementation, it is recommended to provide verification evidence of safety and reliability of the Ada run-time system as well as for the application-specific components. The run-time system that is needed to implement the dynamic semantics of the full Ada concurrency model is complex, and the number of states that can be represented by its dynamic data structures is large. As a result, it is very challenging for a commercial Ada vendor to produce certification evidence to the highest integrity levels for an entire Ada run-time system.

The Ravenscar profile definition greatly reduces the size and complexity of the run-time system, to simplify the process of providing evidence of its safety and reliability. Ada concurrency features that have major impact on the run-time system semantics, such as abort, asynchronous transfer of control, multiple entry queues each with a list of waiting tasks, requeue statements, task hierarchy and dependency, and finalization actions of local protected objects, are eliminated. As a result, it is possible to create not only a small and highly efficient run-time system implementation, but also one that is amenable to the forms of verification applicable to sequential code as described in ISO/IEC TR 15942, which can then be used as evidence to support the formal certification of an entire software system to the highest integrity levels.

5 The Ravenscar profile definition

5.1 Background

The 8th International Real-Time Ada Workshop (IRTAW) was held in April 1997 in Ravenscar, UK. Two position papers[7-8] led to an extended discussion on tasking profiles. By the end of the workshop, the Ravenscar profile had been defined[10] in a form that is almost identical to its current specification.

At the 9th IRTAW[11] (March 1999), the Ravenscar profile was again discussed at length. The definition was reaffirmed and clarified. The most significant change was the incorporation of Suspension Objects. An Ada Letters paper[10] became the de facto defining statement of the Ravenscar profile.

By the 10th IRTAW[12] (September 2000), many of the position papers were on aspects of the Ravenscar profile and its use and implementation. No major changes were made, although an attempt to standardize on the restriction identifiers was undertaken.

At the 11th IRTAW[13] (April 2002), the formal definition of the profile as formulated by the Ada Rapporteur Group (ARG) was agreed. It was confirmed that the Ravenscar profile requires task dispatching policy FIFO_Within_Priorities and locking policy Ceiling_Locking.

Since 2002, the Ravenscar profile has been a formal part of the definition of Ada. Each time the language is upgraded, the profile is revisited to make sure that it continues to have the right set of restrictions. The series of IRTAW workshops continues to review the Ravenscar profile's definition. The last review took place at the 19th IRTAW, in April 2018.

5.2 Definition

The definition of the Ravenscar profile is now included in ISO/IEC 8652:2023 (the Ada Standard). The definition is reported here for information only.

An application requests the use of the Ravenscar profile by means of the configuration pragma Profile with the Ravenscar identifier:

```
pragma Profile(Ravenscar);
```

There are, in general, two distinct ways of defining the details of a profile: either by defining what is in it, or by declaring those parts of Ada that are not. The official definition specifies the restrictions that are needed to reduce the full tasking model to Ravenscar. However, this gives a rather negative definition. Therefore, this subclause introduces the profile by focusing on the features it does contain.

Following from the discussion on verification in 4.5, an adequate set of tasking features is defined. The Ravenscar profile allows programs to contain:

— Task types and objects, defined at the library level.

— Protected types and objects, defined at the library level, with a maximum of one entry per object and with a maximum of one task queued at any time on that entry. The entry barrier can only be a single Boolean variable (or a Boolean literal).

— Atomic and Volatile aspects.

— Delay_until statements.

— Ceiling_Locking policy and FIFO_Within_Priorities dispatching policy.

— The E'Count attribute for protected entries except within entry barriers.

— The Ada.Task_Identification package plus task attributes T'Identity and E'Caller.

— Synchronous task control.

— Task type and protected type discriminants.

— The Ada.Real_Time package.

— Protected procedures as statically bound interrupt handlers.

— Static allocation of task to cores on a multicore (or multiprocessor) platform so that each core hosts a separate set of tasks, to which the Ravenscar profile's scheduling and locking policies apply locally.

Together, these form a coherent set of features that define an adequate language for expressing the programming needs of statically defined real-time systems.

5.3 Summary of implications of pragma Profile (Ravenscar)

The following restrictions apply to the alternative mode of operation defined by the Ravenscar profile. Some restrictions require language features to be omitted, others can be achieved by simply requiring that certain defined (standard) library packages are not incorporated into the program that is conforming to the Ravenscar profile (i.e. there is no semantic dependency on the specified package).

The profile is defined as follows (see ISO/IEC 8652:2023, D.13):

```
pragma Task_Dispatching_Policy(FIFO_Within_Priorities);
pragma Locking_Policy(Ceiling_Locking);
pragma Detect_Blocking;
pragma Restrictions(
                  No_Abort_Statements,
                  No_Dynamic_Attachment,
                  No_Dynamic_CPU_Assignment,
                  No_Dynamic_Priorities,
                  No_Implicit_Heap_Allocations,
                  No_Local_Protected_Objects,
                  No_Local_Timing_Events,
                  No_Protected_Type_Allocators,
                  No_Relative_Delay,
                  No_Requeue_Statements,
                  No_Select_Statements,
                  No_Specific_Termination_Handlers,
                  No_Task_Allocators,
                  No_Task_Hierarchy,
                  No_Task_Termination,
                  Simple_Barriers,
                  Max_Entry_Queue_Length => 1,
                  Max_Protected_Entries => 1,
                  Max_Task_Entries => 0,
                  No_Dependence => Ada.Asynchronous_Task_Control,
                  No_Dependence => Ada.Calendar,
                  No_Dependence => Ada.Execution_Time.Group_Budgets,
                  No_Dependence => Ada.Execution_Time.Timers,
                  No_Dependence => Ada.Synchronous_Barriers,
                  No_Dependence => Ada.Task_Attributes,
                  No_Dependence => System.Multiprocessors.Dispatching_Domains);
```

6 Rationale

6.1 General

This clause provides a description of each restriction, a detailed rationale for the imposition of each restriction and some general discussion about how to work within the restrictions while still retaining flexibility in the design and coding processes.

6.2 Ravenscar profile restrictions

6.2.1 Static existence model

The restrictions listed below ensure that the set of tasks and interrupts to be analysed is fixed and has static properties (in particular, base priority) after program elaboration. If a variable task set were to exist, then it

would be impractical to perform static timing analysis of the program because of the dynamic nature of the requirements for execution time and the meeting of deadlines.

No_Task_Hierarchy

(see ISO/IEC 8652:2023, D.7) No task depends on a master other than the library-level master.

The restriction No_Task_Hierarchy prevents the declaration of tasks local to procedures or to other tasks. Thus, tasks may only be created at the library level, i.e. within the declarative part of library level package specifications and bodies, including child packages and package subunits.

No_Task_Allocators

(see ISO/IEC 8652:2023, D.7) There are no allocators for task types or types containing task subcomponents.

The restriction No_Task_Allocators prevents the dynamic creation of tasks via the execution of Ada allocators (see ISO/IEC 8652:2023, 4.8)

No_Task_Termination

(see ISO/IEC 8652:2023, D.7) All tasks are non-terminating. It is implementation-defined what happens if a task attempts to terminate. If there is a fall-back handler set for the partition it should be called when the first task attempts to terminate.

The restriction attempts to mitigate the hazard that can be caused by tasks terminating silently. Real-time tasks normally have an infinite loop as their last outermost statement.

No_Specific_Termination_Handlers

(see ISO/IEC 8652:2023, D.7) There is no use of a name denoting the Set_Specific_Handler and Specific_Handler subprograms in Task_Termination.

The restriction No_Specific_Termination_Handlers ensures that the only termination handler defined for the program is a fall-back handler (see ISO/IEC 8652:2023, C.7.3).

No_Abort_Statements

(see ISO/IEC 8652:2023, D.7) There are no abort_statements, and there is no use of a name denoting Task_Identification.Abort_Task.

The restriction No_Abort_Statements ensures that tasks cannot be aborted. The removal of abort statements (and select then abort) significantly reduces the size and complexity of the run-time system. It also reduces non-determinacy.

No_Dynamic_Attachment

(see ISO/IEC 8652:2023, D.7) There is no use of a name denoting any of the operations defined in package Interrupts (Is_Reserved, Is_Attached, Current_Handler, Attach_Handler, Exchange_Handler, Detach_Handler, and Reference).

The restriction No_Dynamic_Attachment excludes use of the operations in predefined package Ada.Interrupts, which contains primitives to attach and detach handlers dynamically during program execution. In conjunction with restriction No_Local_Protected_Objects (see below) this implies that interrupt handlers can only be attached statically using Attach_Handler applying to protected procedures within library-level protected objects. Note the types and names defined in Ada.Interrupts can be used.

No_Dynamic_Priorities

(see ISO/IEC 8652:2023, D.7) There are no semantic dependencies on the package Ada.Dynamic_Priorities, and no occurrences of the attribute Priority.

The restriction No_Dynamic_Priorities disallows the use of the predefined package Ada.Dynamic_Priorities, thereby ensuring that the priority assigned at task creation is unchanged during task execution, except when the task is executing a protected operation, during which time it inherits the ceiling priority. Protected objects also have unchanging ceiling priorities (this follows from the restriction to not use the Priority attribute (see ISO/IEC 8652:2023, 4.1.4).

No_Local_Timing_Events

(see ISO/IEC 8652:2023, D.7) Timing events are declared only at library level.

The restriction No_Local_Timing_Events prevents the declaration of timing events local to procedures or tasks. Thus, Timing_Events may only be created at the library level.

6.2.2 Static synchronization and communication model

The restrictions listed below are a natural consequence of the static existence model, since a locally declared protected object is meaningless for mutual exclusion and task synchronization purposes if it can only be accessed by one task. Furthermore, a static set of protected objects should be used to render the program amenable to schedulability analysis.

No_Local_Protected_Objects

(see ISO/IEC 8652:2023, D.7) Protected objects are declared only at library-level.

The restriction No_Local_Protected_Objects prevents the declaration of protected objects local to subprograms, tasks, or other protected objects.

No_Protected_Type_Allocators

(see ISO/IEC 8652:2023, D.7) There are no allocators for protected types or types containing protected type subcomponents.

The restriction No_Protected_Type_Allocators prevents the dynamic creation of protected objects via Ada allocators (see ISO/IEC 8652:2023, 4.8).

No_Select_Statements

(see ISO/IEC 8652:2023, D.7) There are no `select_statements`.

Max_Task_Entries = > N

(see ISO/IEC 8652:2023, D.7) Specifies the maximum number of entries per task.

For the Ravenscar profile, the value of Max_Task_Entries is zero.

The restrictions Max_Task_Entries = > 0 and No_Select_Statements prohibit the use of Ada rendezvous for task synchronization and communication. This ensures that these operations are achieved using only the two supported task synchronization primitives: protected object entries and suspension objects, both of which exhibit the time-deterministic execution properties needed for static timing analysis.

6.2.3 Deterministic memory usage

The Ravenscar profile contains two restrictions that are designed to prevent implicit dynamic memory allocation by the implementation. The Ravenscar profile does not prevent the use of the standard storage pool or a user-defined storage pool via explicit allocators. However, if there is no application-level visibility or control over how the storage in the standard storage pool is managed, the use of this pool is not be recommended.

No_Implicit_Heap_Allocations

(see ISO/IEC 8652:2023, D.7) There are no operations that implicitly require heap storage allocation to be performed by the implementation. The operations that implicitly require heap storage allocation are implementation defined.

The restriction No_Implicit_Heap_Allocations prevents the implementation from allocating memory from the standard storage pool other than as part of the execution of an Ada allocator.

No dependence on Ada.Task_Attributes

(see ISO/IEC 8652:2023, C.7.2) There are no semantic dependencies on the package Ada.Task_Attributes.

The restriction No_Task_Attributes_Package prevents use of the predefined package Ada.Task_Attributes (see ISO/IEC 8652:2023, C.7.2), which is used to dynamically create attributes of each task in the application. Attribute creation can cause implicit dynamic allocation of memory. Although an implementation is allowed to statically reserve space for such attributes and then to impose a restriction on usage, it is felt that support of this feature is not compatible with the static nature of Ravenscar programs.

6.2.4 Deterministic execution model

The following restrictions ensure deterministic execution:

Max_Protected_Entries = > N

(see ISO/IEC 8652:2023, D.7) Specifies the maximum number of entries per protected type. The bounds of every entry family of a protected unit are static or defined by a discriminant of a subtype whose corresponding bound is static.

For the Ravenscar profile, the value of Max_Protected_Entries is 1.

Max_Entry_Queue_Length = > N

(see ISO/IEC 8652:2023, D.7) Defines the maximum number of calls that are queued on an entry. Violation of this restriction results in the raising of Program_Error exception at the point of the call.

For the Ravenscar profile, the value of Max_Entry_Queue_Length is 1, and a call can only be queued on a protected entry, since Max_Task_Entries is 0.

Restrictions Max_Protected_Entries = > 1 and Max_Entry_Queue_Length = > 1 ensure that at most one task can be suspended waiting on a closed entry barrier for each protected object which is used as a task synchronization primitive. This avoids the possibility of queues of task calls forming on an entry, with the associated non-determinism of the length of the waiting time in the queue. It also avoids two or more barriers becoming open simultaneously as the result of a protected action, with the associated non-determinism of selecting which entry should be serviced first. The restriction also enables a tight time bound on the epilogue code to be determined.

The Max_Entry_Queue_Length restriction can only be checkable at run time, in which case violation would result in the raising of the Program_Error exception at the point of the entry call. This is consistent with the Ada rule that states that Program_Error exception is raised upon calling Suspend_Until_True if another task is waiting on that suspension object (when pragma Detect_Blocking is enabled as it is in the Ravenscar profile) (see ISO/IEC 8652:2023, D.10). An application can further restrict a Ravenscar program so that only one task is able to call the specific entry. A static check can then be provided, but this goes beyond what the Ravenscar profile defines.

When the restriction Max_Entry_Queue_Length = > 1 is in force, Queuing_Policy (see ISO/IEC 8652:2023, D.4) has no effect, since there are no queues.

Simple_Barriers

(see ISO/IEC 8652:2023, D.7) The Boolean expression in an entry barrier is either a static expression or a name that statically denotes a component of the enclosing protected object.

The restriction Simple_Barriers, coupled with Max_Protected_Entries = > 1, ensures a deterministic execution time and absence of side effects for the evaluation of entry barriers at the epilogue of protected actions within a protected object that is used for task synchronization. There is also scope for additional optimization by the implementation since the barrier value is either static or can be read directly from one of the protected object components, without needing to be computed separately. If the application requires composite entry barrier expressions, this can be achieved by declaring an additional Boolean in the protected data and assigning the composite expression to the Boolean whenever its evaluation result can change. The Boolean variable used for the barrier expression is declared within the protected object (or type).

No_Requeue_Statements

(see ISO/IEC 8652:2023, D.7) There are no `requeue_statements`.

The restriction No_Requeue_Statements ensures deterministic task release from protected entry barriers used for task synchronization. The `requeue_statement` in Ada causes the current caller of a protected entry to be requeued to a different entry dynamically, thereby making it difficult to perform static analysis of task release.

No dependence on Ada.Asynchronous_Task_Control

(see ISO/IEC 8652:2023, D.13) There are no semantic dependencies on the package Ada.Asynchronous_Task_Control.

The restriction No_Asynchronous_Control excludes the use of asynchronous suspension of execution. This ensures that task execution is temporally deterministic. See also the comments made on restriction No_Abort_Statements in 6.2.1.

No_Relative_Delay

(see ISO/IEC 8652:2023, D.7) There are no `delay_relative_statements`, and there is no use of a name that denotes the Timing_Events.Set_Handler subprogram that has a Time_Span parameter.

The restriction No_Relative_Delay prohibits use of the `delay_relative_statement` based on type Duration. This statement exhibits non-determinism with respect to the absolute time at which the delay expires in the case when the delaying task is pre-empted after calculating the specified relative delay, but before actual suspension occurs. In contrast, the `delay_until_statement` is deterministic and should be used for accurate release of time-triggered tasks.

No dependency on Ada.Calendar

(see ISO/IEC 8652:2023, D.13) There are no semantic dependencies on the package Ada.Calendar.

The restriction No_Calendar ensures that all timing is performed using the high precision afforded by the time type in package Ada.Real_Time (see ISO/IEC 8652:2023, D.8), or by an implementation-defined time type. The Ada.Real_Time time type has a precision of the same order of magnitude as the real-time clock device on the underlying processor board. In contrast, the time type in package Calendar generally has much coarser precision than the real-time clock, due to the need to support a 200-year range and can jump backward or forward in the event of calendar adjustments (e.g., leap years, leap seconds). Its use can thus result in less accuracy in task release times.

6.2.5 Simple run-time behaviour

To reduce the overheads associated with supporting the full Ada model, some features are removed from the Ravenscar profile, in particular, time-triggered tasks.

No dependency on Ada.Execution_Time.Group_Budgets

(see ISO/IEC 8652:2023, D.13) There are no semantic dependencies on the package Ada.Execution_Time.Group_Budgets.

A Ravenscar runtime can monitor the execution time of tasks, but it does not support the sharing of an execution-time budget within a group of tasks. Neither does it require a handler to be executed if a task executes beyond a defined level of execution time (hence the next restriction). This simplifies the runtime but makes it harder to construct programs that can recover from timing errors.

No dependency on Ada.Execution_Time.Timers

(see ISO/IEC 8652:2023, D.13) There are no semantic dependencies on the package Ada.Execution_Time.Timers.

6.2.6 Parallel semantics

More recent definitions of the Ada language have included features that provide more control over the execution of multi-tasking programs on parallel hardware. Such hardware includes multiprocessors (with various memory configurations), multi-core processor and various forms of heterogeneous architectures. The definition of the Ravenscar profile has been extended to deal with these forms of truly parallel (rather than just concurrent) execution. The basic approach chosen for the Ravenscar profile has been to support the

static allocation of tasks to processors. Conformance with this choice is achieved by applying the following restrictions:

No_Dynamic_CPU_assignment

The abbreviation CPU stands for Central Processing Unit, and can be understood as “processor”. All of the tasks in the partition will execute on the same processor, unless the programmer explicitly uses aspect CPU to specify the task-to-processor assignment (see ISO/IEC 8652:2023, D.13).

This results in tasks being statically assigned to processors.

No dependency on Ada.Synchronous_Barriers

(see ISO/IEC 8652:2023, D.13) There are no semantic dependencies on the package Ada.Synchronous_Barriers.

Synchronous barriers (see ISO/IEC 8652:2023, D.10.1) are used on some forms of parallel hardware. As they can be programmed by the user in a Ravenscar application, the use of the predefined package is not explicitly supported by the Ravenscar profile.

No dependency on System.Multiprocessors.Dispatching_Domains

(see ISO/IEC 8652:2023, D.13) There are no semantic dependencies on the package System.Multiprocessors.Dispatching_Domains.

Dispatching domains allow more structured approaches to parallel execution to be supported. Currently, this leads to programs that are deemed to be beyond what can be easily analysed; they are therefore not included in the Ravenscar profile.

6.2.7 Implicit restrictions

The set of restriction identifiers for Ada does not represent an orthogonal set of restrictions with the result that some restrictions are implied by others. For example, restriction No_Select_Statements implies restriction Max_Select_Alternatives will be set to zero.

6.3 Ravenscar profile dynamic semantics

6.3.1 Task dispatching policy

The task dispatching policy that is associated with pragma Profile(Ravenscar) is set to FIFO_Within_Priorities (see ISO/IEC 8652:2023, D.2).

6.3.2 Locking policy

The locking policy that goes with pragma Profile(Ravenscar) is set to Ceiling_Locking (see ISO/IEC 8652:2023, D.3). This policy provides one of the lowest worst-case blocking times for contention for shared resources, and therefore maximizes the schedulability of the task set when pre-emptive scheduling is used

6.3.3 Queuing policy

The queuing policy is not meaningful for pragma Profile(Ravenscar) since no entry queues can form. Thus, queuing policy identifiers FIFO_Queuing and Priority_Queuing have no effect.

6.3.4 Additional run-time errors defined by the Ravenscar profile

ISO/IEC 8652:2023 (the Ada Standard) defines a number of concurrency-related run-time checks that can lead to the raising of an exception. The Ravenscar profile restrictions greatly reduce the quantity of these checks, and thus the number of exception cases that can occur. The two concurrency-related run-time checks that apply to Ravenscar programs are:

a) detection of priority ceiling violation as defined by Ceiling_Locking policy;

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

The Ravenscar profile introduces some additional concurrency-related checks that are potentially detectable only at execution time:

c) the maximum number of calls that are queued concurrently on an entry is not allowed to exceed one. Program_Error exception is raised if the error occurs [pragma Restrictions(Max_Entry_Queue_Length = > 1)];

d) all tasks are non-terminating [pragma Restrictions(No_Task_Termination)].

A conforming implementation documents the effect of a task that attempts to terminate. Possible effects can include:

e) allowing the task to terminate silently;

f) holding the task in a permanent pre-terminated state;

g) executing a task termination handler.

Whatever action is taken by the implementation, the application cannot assume that full task termination actions (including finalization) have been executed.

6.3.5 Potentially-blocking operations in protected actions

The Ravenscar profile requires detection of the following bounded error with the consequential raising of Program_Error exception:

— execution of a potentially-blocking operation during a protected action (pragma Detect_Blocking).

The Ravenscar profile definition does however significantly reduce the list of potentially-blocking operations that can occur during a protected action. In particular, the following potentially-blocking operations are eliminated by the Ravenscar profile definition:

— a `select_statement`

— an `accept_statement`

— a task entry call

— a `delay_relative_statement`

— an `abort_statement`

— task creation or activation

— an external requeue_statement with the same target object as that of the protected action.

The Ravenscar profile definition does not require detection, at compile time, of other potentially blocking operations defined by the language standard (see ISO/IEC 8652:2023, 9.5.1). In this case, the detection may occur at the point of execution of the potentially blocking operation within the called subprogram body.

The rationale for requiring detection of potentially-blocking operations in protected actions is to allow a highly efficient and temporally deterministic implementation of Ceiling_Locking policy on a mono-

processor. In effect, the ceiling priority alone is sufficient to provide mutual exclusion without the need to use locks such as mutexes once it is guaranteed that the task cannot suspend co-operatively while inside the protected operation. This form of locking is also non-queuing on a mono-processor, with the associated benefit of removing the need to compute the worst-case duration that a task call can wait in the queue.

6.3.6 Exceptions and the No_Exceptions restriction

The general concern within high-integrity systems of the occurrence of unhandled exceptions is not addressed directly by the Ravenscar profile since exceptions relate to the sequential, rather than the concurrent, part of the language. Consequently, 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 can go undetected. In particular, an unhandled exception can cause any of the following effects:

— silent abandonment of the execution of an interrupt handler;

— silent termination of a task;

— premature exit from a protected action.

The Ravenscar profile statically avoids the possibility that an exception can be raised by an entry barrier via the restriction Simple_Barriers. In addition, the Ravenscar profile imposes the restriction No_Task_Termination that requires the implementation to document the effect of a task attempting to terminate. Nevertheless, this is inadequate for most high integrity applications that require static demonstration of absence of exceptions due to run-time check failure. Some techniques are presented in 8.3.3 to address the topic of proof of absence of the concurrency-related run-time errors that can occur in a Ravenscar profile program, using static analysis.

ISO/IEC 8652 includes the identifier No_Exceptions as a valid argument for the Restrictions pragma. It should be noted that the inclusion of this pragma does not provide a static guarantee of exception freedom – it merely guarantees that the application code does not contain any explicit `raise_statement`, nor code generation for language-defined checks, nor any exception handlers. However, it is possible for an exception to be raised automatically by the underlying hardware, or by built-in code in the run-time system. There is a documentation requirement on the implementation to define such cases (see ISO/IEC 8652:2023, H.4).

In addition, the language standard defines execution of a program to become erroneous if a language-defined check is suppressed via restriction No_Exceptions and the conditions arise that would have caused the check to fail (see ISO/IEC 8652:2023, H.4). This is consistent with the suppression of checks using `pragma Suppress` (see ISO/IEC 8652:2023, 11.15). Since erroneous execution results in the behaviour of a program becoming undefined, the recommendation for high integrity systems is that restriction No_Exceptions should only be used in conjunction with verification and analysis techniques that can statically prove that no exceptions due to run-time check failure can occur. In this case, restriction No_Exceptions is providing the additional safeguard that exception raising via explicit `raise_statement`s will be prohibited at compile time.

6.3.7 Access to shared variables

The Ravenscar profile requires all synchronization and communication between tasks and interrupt handlers to use data that are guaranteed to have mutually exclusive access. This prevents any erroneous execution that can arise if concurrent access (that includes a write operation) to the same unprotected shared variable is permitted. Such access control is provided in Ada using one of the following constructs:

— a protected object;

— a suspension object;

— an atomic object (to which the Atomic aspect applies).

This access control model applies to the operational phase of the application, after program initialization via elaboration of library-level packages is complete. For each class of object above, it is possible to ensure that its initialization is completed as part of program elaboration.

There is an issue however, in that the semantics of Ada define task activation and interrupt handler attachment to occur during library-level elaboration code for objects that are declared within library-level packages. Consequently, it is the case that tasks will execute their declarative part and can proceed into their `sequence_of_statements`, and that interrupt handlers can execute, prior to the elaboration code for program initialization being completed. This scenario can give rise to the following undesirable effects:

— a task body or interrupt handler can suffer an access-before-elaboration exception;

— a task body or interrupt handler can access uninitialized data;

— a task body or interrupt handler can access unprotected data concurrently that it shares only with the thread of control that is performing the data initialization.

It is possible to program each task such that it suspends itself at the start of its sequence of statements, but this is not possible for interrupt handlers (although an application can inhibit interrupts if the device allows). Furthermore, the code executed as part of task activation (prior to the suspension point) can suffer the effects listed above. In order to address this issue, the Partition_Elaboration_Policy is defined in ISO/IEC 8652:2023 (see 6.3.8).

6.3.8 Elaboration control

The new pragma Partition_Elaboration_Policy (see ISO/IEC 8652:2023, H.6) is not part of the Ravenscar profile, but it is closely related to it. If given the argument Sequential, this defines an alternative elaboration behaviour in which all tasks declared at the library level proceed to their activation only after the environment task has completed all its elaborations and the main program is leaving its `declarative_part`. It is only at that point that interrupt handlers are attached (so that no interrupt can be delivered earlier), and all tasks eventually start their concurrent execution. This pragma complements those that are defined by the Ravenscar profile and helps achieve the goal that controlled access to global shared variables is met during program initialization.

7 Examples of use

7.1 General

This clause illustrates some simple patterns of use of the Ravenscar profile.

The Ravenscar profile can be used with a variety of coding styles. However, if the user performs program analysis, for example to check the schedulability of the tasks, then the adoption of certain coding styles is recommended. Indeed, a small number of templates can cater for a large class of application needs. In 7.2 to 7.9, examples are given to illustrate the straightforward use of the Ravenscar profile. Thereafter, in 7.10 to 7.13, examples show how the Ravenscar profile can deal with requirements that would appear to lie outside of its scope.

To conform with the Ada language specification as per ISO/IEC 8652:2023, aspects should be used in place of most pragmas. Accordingly, all occurrences of the obsolete pragmas have been replaced with the corresponding `aspect`.

7.2 Cyclic task

The task body for a cyclic (or periodic) task typically has, as its last statement, an outermost infinite loop containing one or more `delay_until_statements`. The basic form of a cyclic task has just a single delay statement either at the start or at the end of the statements within the loop. The Ravenscar profile supports only one “time” type for use as the argument – Ada.Real_Time.Time, although a user-defined time type can be used.

Task termination is considered an error condition in Ravenscar-compliant code since there is no dynamic task creation (and hence the thread of control would be permanently lost). Hence, the loop that is presented in the template below is infinite.

A cyclic task will not usually contain any other form of voluntary-suspension statement in the infinite loop, since this would undermine the schedulability analysis.

The Ravenscar profile supports the use of discriminants for task types and protected types. One use of a discriminant is to set differing priorities for task objects or protected objects that are of the same type by using it as the argument of the Priority aspect.

Discriminants can also be used to indicate the period of a cyclic task or other task parameters, including the assigned priority.

EXAMPLE Cyclic Template

```
task type Cyclic(Pri: System.Priority; Cycle_Time: Positive)
   with Priority => Pri;

task body Cyclic is
   Next_Period: Ada.Real_Time.Time;
   Period: constant Ada.Real_Time.Time_Span:=
                            Ada.Real_Time.Microseconds(Cycle_Time);
   -- Other declarations as needed
begin
   -- Initialization code
   Next_Period:= Ada.Real_Time.Clock + Period;
   loop
      delay until Next_Period; -- Wait one whole period before executing
      -- Non-suspending periodic response code
      -- May include calls to protected procedures
      Next_Period:= Next_Period + Period;
   end loop;
end Cyclic;

-- Declare two task objects of this type
C1: Cyclic(20,200);
C2: Cyclic(15,100);
```

Cyclic tasks normally exchange data through protected operations. In this coding style, there are no protected entries since the only activation event is on `delay until`. Conformance with the Ravenscar profile involves all shared data being placed in protected objects to avoid corruption. As noted in 4.2.3, atomic objects can also be used to help share objects safely, together with a read/write protocol statically proven free of race conditions.

7.3 Coordinated release of cyclic tasks

The simple example illustrated in 7.2 has two cyclic tasks that read the clock and then suspend for the time duration specific in the constant "Period". It can however by useful for all such tasks to coordinate their start times so that they share a common epoch. This can help to enforce precedence relations across tasks. To achieve this, a protected object is used, which reads the clock on creation and then makes this clock value available to all cyclic tasks.

EXAMPLE 1 Protected object implementing an Epoch

```
protected Epoch
   with Priority => System.Priority'Last
is
   function Start_Time return Ada.Real_Time.Time;
private
   Start: Ada.Real_Time.Time:= Ada.Real_Time.Clock;
end Epoch;

protected body Epoch is
   function Start_Time return Ada.Real_Time.Time is
   begin
      return Start;
   end Start_Time;
end Epoch;
```

A protected object is not strictly needed to this end, since a shared variable appropriately initialized will suffice. A more robust scheme, which only reads the epoch time once a task actually needs it, is as follows.

EXAMPLE 2 Caller initialized Epoch

```
protected Epoch
   with Priority => System.Priority'Last
is
   procedure Get_Start_Time(T: out Ada.Real_Time.Time);
private
   Start: Ada.Real_Time.Time;
   First: Boolean:= True;
end Epoch;

protected body Epoch is
   procedure Get_Start_Time(T: out Ada.Real_Time.Time) is
   begin
      if First then
        First:= False;
        Start:= Ada.Real_Time.Clock;
      end if;
      T:= Start;
   end Get_Start_Time;
end Epoch;
```

This leads to the following further example.

EXAMPLE 3 Cyclic task with Offsets

```
task type Cyclic(Pri: System.Priority; Cycle_Time: Positive)
   with Priority => Pri;

task body Cyclic is
   Next_Period: Ada.Real_Time.Time;
   Period: constant Ada.Real_Time.Time_Span:=
                                    Ada.Real_Time.Microseconds(Cycle_Time);

   -- Other declarations as needed
begin
   -- Initialization code
   Epoch.Get_Start_Time(Next_Period);
   Next_Period:= Next_Period + Period;
   loop
      delay until Next_Period;   -- Wait until next period after epoch
      -- Non-suspending periodic response code
      -- May include calls to protected procedures
      Next_Period:= Next_Period + Period;
   end loop;
end Cyclic;
```

7.4 Cyclic tasks with precedence relations

The use of priorities and a shared epoch can be used to enforce precedence between tasks with the same period, given that the application can be restricted so that the tasks do not block during execution. An alternative scheme is to use an offset in time. Here, scheduling analysis is used to ensure that each task has been completed before the next task is released.

EXAMPLE Cyclic task using Epoch

```
task type Cyclic(Pri: System.Priority;
                 Cycle_Time, Offset: Natural)
   with Priority => Pri;

task body Cyclic is
   Next_Period: Ada.Real_Time.Time;
   Period: constant Ada.Real_Time.Time_Span:=
                                 Ada.Real_Time.Microseconds(Cycle_Time);
   -- Other declarations
```

```
begin
   -- Initialization code
   Next_Period:= Epoch.Start_Time + Ada.Real_Time.Microseconds(Offset);
   loop
      delay until Next_Period;  -- Wait until next period after offset
      -- Non-suspending periodic response code
      -- May include calls to protected procedures
      Next_Period:= Next_Period + Period;
   end loop;
end Cyclic;

First: Cyclic(20,200,0);
Second: Cyclic(20,200,70);  -- Required to complete with deadline 70
```

7.5 Event-triggered tasks

The task body for an event-triggered task that conforms to the Ravenscar profile typically has, as its last statement, an outermost infinite loop whose first statement is either a call to a protected entry or a call to Ada.Synchronous_Task_Control.Suspend_Until_True using a Suspension Object. The suspension object is used when no other effect is required in the signalling operation; for example, no data are to be transferred from signaller to waiter. In contrast, the protected entry is used for more elaborate event signalling, when additional operations accompany the resumption of the event-triggered task.

An event-triggered task will not usually contain any other form of voluntary-suspension statement in the infinite loop.

EXAMPLE An event-triggered task

```
-- A suspension object, SO, is declared in a visible library unit and is
-- set to True in another (releasing) task

task type Sporadic(Pri: System.Priority)
   with Priority => Pri;

task body Sporadic is
   -- Declarations
begin
   -- Initialization code
   loop
      Ada.Synchronous_Task_Control.Suspend_Until_True(SO);
      -- Non-suspending sporadic response code
   end loop;
end Sporadic;

Sp: Sporadic(17);
```

7.6 Shared resource control using protected objects

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 only for task synchronization purposes where data exchange is involved. A protected procedure should be used when the internal state of the protected data are intended to be altered, and a protected function should be used for information retrieval from the protected data, when the data remains unchanged.

ISO/IEC 8652:2023, 9.5.1, states that the use of any form of voluntary-suspension statement during the execution of a protected operation is a bounded error. The Ravenscar profile requires, via pragma Detect_Blocking, an implementation to detect this error (and hence to raise the Program_Error exception), other than in the case when suspension is due to execution outside of the Ada environment, for example within an underlying operating system call or within imported code that is written in another language.

It is essential to choose the correct value for the ceiling priority of the protected object. By default, the value is System.Priority'Last, unless the protected object contains interrupt handlers (see below). The chosen value is recommended to be at least as high as the highest priority task that calls one of the protected operations. If this is not the case, the Ada Reference Manual requires the Program_Error exception to be raised when a task with a priority higher than the ceiling priority makes a call to one of the protected operations. However,

if the ceiling value is higher than necessary, there can be an increase in the blocking time that high priority tasks will suffer, and consequently a decrease in the overall schedulability of the system. Tool support can be available to determine the optimal ceiling value when the calling sequences can be statically analysed.

EXAMPLE Use of protected object for mutual exclusion

```
protected Shared_Data
   with Priority => 10  -- All callers will have priority no greater than 10
is
   function Get return Data;  -- For some global type, Data
   procedure Put(D: in Data);
private
   Current: Data; -- 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;
```

7.7 Task synchronization primitives

Task synchronization, in the form of a wait/signal event model, can be achieved in the Ravenscar profile using either a protected entry or a suspension object, as described above for event-triggered tasks.

The suspension object is the optimized form for a simple suspend/resume operation. The package Ada.Synchronous_Task_Control (see ISO/IEC 8652:2023, D.10) is used to declare a suspension object, and the primitives Suspend_Until_True and Set_True are used for the suspend and resume operations respectively.

The use of protected objects with entries for task synchronization is restricted by the Ravenscar profile. The protected object can have a maximum of one entry declaration. The entry barrier is recommended to be a simple value that is either a Boolean literal or a Boolean variable that is part of the protected state. A maximum of one task is allowed to wait on the protected entry at any time (see 6.2.4). These restrictions provide the necessary determinism in knowing which waiting task is serviced first when entry barriers become true, since there can be at most one such task call enqueued at it. This model is very similar to the suspension object approach except that:

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

Additional code can be executed atomically as part of signalling by use of the bodies of the protected operations.

EXAMPLE Event-Triggered Tasks Suspending on a Protected Entry

```
protected type Event(Ceiling: System.Priority)
   with Priority => Ceiling --  Ceiling priority defined for each object
is
   entry Wait(D: out Data);
   procedure Signal(D: in Data);
private
   Current: Data;  -- Event data declaration
   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;

Event_Object: Event(15);

task Event_Handler
   with Priority => 14; --  I.e. this must be not greater than 15

task body Event_Handler is
   -- Declarations, including D of type Data
begin
   -- Initialization code
   loop
     Event_Object.Wait(D);
     -- Non-suspending event handling code
   end loop;
end Event_Handler;
```

7.8 Minimum separation between event-triggered tasks

To ensure the timely execution of all tasks in a system, it can be necessary to enforce a separation between sporadic tasks so that they cannot execute more frequently than some agreed value. This is easily achieved with a `delay_until_statement`. Doing so however, introduces a second activation event into the code of the task's outer loop. In general, this can make the task more difficult to analyse. In the following example however, it actually facilitates the analysis by ensuring a minimum separation between task activations. This happens because the two activation events are in effect subsequent.

EXAMPLE Event-triggered task with minimum separation

```
task Event_Handler
   with Priority(14);

task body Event_Handler is
   -- Declarations, including D of type Data
   Minimum_Separation: constant Ada.Real_Time.Time_Span:=
                                        -- some appropriate value
   Next: Ada.Real_Time.Time;
begin
   -- Initialization code
   loop
      Event_Object.Wait(D);
      Next:= Ada.Real_Time.Clock + Minimum_Separation;
      -- Non-suspending event handling code
      delay until Next;  -- this ensures minimum temporal separation
   end loop;
end Event_Handler;
```

7.9 Interrupt handlers

The code of an interrupt handler will often be used to initiate a response in an event-triggered task. This is because the code in the handler itself executes at the hardware interrupt level, and typically the major part of the processing of the response to the interrupt is moved into an event response task, which executes at a software priority level with interrupts fully enabled.

In the previous example, if signalling is intended to be achieved via an interrupt, then the procedure Signal should be defined as parameterless, and be identified as an interrupt handler by the aspect Attach_Handler. This aspect includes an argument of type Ada.Interrupts.Interrupt_ID that identifies the interrupt to which the handler applies.

The ceiling priority of a protected object that contains an interrupt handler is recommended to be in the range of System.Interrupt_Priority.

EXAMPLE Interrupt handling via a protected entry

```
protected Interrupt_Event
   with Interrupt_Priority => System.Interrupt_Priority'Last
is
   entry Wait(D: out Data);
   procedure Signal
      -- Must be parameterless
      with Attach_Handler => Some_Interrupt_Id;
   -- Wait and Signal will execute with full interrupt lockout
private
   Current: Data;  -- Event data declaration
   Signalled: Boolean:= False;
end Interrupt_Event;

protected body Interrupt_Event is -- Similar to the code in the previous example
   -- except that the setting of Current is obtained via a register during
   -- the execution of Signal rather than as an in parameter
```

7.10 Catering for entries with multiple callers

7.10 to 7.13 illustrate how to cater for situations that appear to need more functionality than provided by the Ravenscar profile. It should not be assumed that the Ravenscar subset is able to deal with all situations that full Ada covers. The tasking features of Ada represent a powerful set of abstractions for programming concurrent and real-time systems. To gain predictability and efficiency, the Ravenscar profile has had to drop many of these features, and it is not appropriate to reintroduce them via a combination of programming tricks and conventions. However, situations can arise when a requirement in just part of a program seems outside of the Ravenscar profile's definition. These can often be catered for by straightforward techniques that benefit from the other restrictions of the Ravenscar profile.

This subclause focuses on the requirement for two (or more) tasks to call the same entry of some protected object. As an illustration, consider a situation in which a series of tasks create work items, while others consume them. For example, if more than 10 outstanding items ever accumulate, then the two separate event-triggered tasks are released. An atomicity requirement is that the two tasks are only released if both are available and only when new work items are created.

EXAMPLE 1 A non Ravenscar Example

```
protected Controller is
   entry Overload;  -- called by two tasks
   procedure Create;
   procedure Consume;
private
   Work_Items: Integer:= 0;
   Released: Boolean:= False;
end Controller;

protected body Controller is
   entry Overload when Released is
   begin
      if Overload'Count = 0 then -- barrier is closed when both tasks have left
         Released:= False;
      end if;
   end Overload;
   procedure Create is
   begin
      Work_Items:= Work_Items + 1;
      Released:= (Work_Items > 10 and Overload'Count = 2);
      -- barrier is opened when more than 10 items and both tasks are waiting
   end Create;
   procedure Consume is
   begin
      Work_Items:= Work_Items - 1;
   end Consume;
end Controller;
```

To conform with the Ravenscar profile restrictions, two Controller protected objects are needed, one for each task. To obtain the required atomicity, the second Controller is called from the first.

EXAMPLE 2 Using multiple protected objects to mimic an entry queue

```
protected First_Controller is
   entry Overload;  -- called by one task
   procedure Check_Called(OK: out Boolean);
private
   Released: Boolean:= False;
end First_Controller;

protected body First_Controller is
   entry Overload when Released is
   begin
      Released:= False; -- barrier set to False once task has been released
   end Overload;
   procedure Check_Called(OK: out Boolean) is
   begin
      Released:= (Overload'Count = 1);
      OK:= Released; -- returns True if task waiting
   end Check_Called;
end First_Controller;

protected Second_Controller is
   entry Overload;  -- called by the other task
   procedure Create;
   procedure Consume;
private
   Work_Items: Integer:= 0;
   Released: Boolean:= False;
end Second_Controller;

protected body Second_Controller is
   entry Overload when Released is
   begin
      Released:= False; -- barrier set to False once task has been released
   end Overload;
   procedure Create is
   begin
      Work_Items:= Work_Items + 1;
      if Work_Items > 10 and Overload'Count = 1 then
         First_Controller.Check_Called(Released);
      end if;  -- if Released is true then the first task has been released
               -- and the second one must also be released
   end Create;
   procedure Consume is
   begin
      Work_Items:= Work_Items - 1;
   end Consume;
end Second_Controller;
```

It should be noted that in the Ravenscar profile, once a task calls an entry, it cannot cancel the call; hence the above algorithm is safe. In the full language, task calls can be cancelled and therefore the above approach is not guaranteed to work.

7.11 Catering for protected objects with more than one entry

To illustrate the way a two-entry protected object can be transformed, consider the standard buffer with one task calling the buffer to extract an item and another task calling it to place items into the buffer. Usually, both of these calls are made via entries in a protected object as the extract call is required to block if the buffer is empty, and the place call is required to block if the buffer is full. To comply with the Ravenscar profile restriction of only one entry in any protected object, a protected object is used for mutual exclusion only and two suspension objects are introduced for the necessary conditional synchronization.

EXAMPLE A bounded buffer example in Ravenscar

```
package Buffer is
   procedure Place_Item(Item: Some_Type);
   procedure Extract_Item(Item: out Some_Type);
end Buffer;

package body Buffer is
   protected Buff is
      procedure Place(Item    : in Some_Type;
                      Success: out Boolean);
      procedure Extract(Item   : out Some_Type;
                      Success: out Boolean);
   private
      Buffer_Full: Boolean:= False;
      Buffer_Empty: Boolean:= True;
      -- other declarations
   end Buff;

   Non_Full, Non_Empty: Ada.Synchronous_Task_Control.Suspension_Object;

   procedure Place_Item(Item: Some_Type) is
      OK: Boolean;
   begin
      Buff.Place(Item, OK);
      if not OK then
         Ada.Synchronous_Task_Control.Suspend_Until_True(Non_Full);
         -- note this is a task activation event
         Buff.Place(Item, OK); -- OK must be true
      end if;
      Ada.Synchronous_Task_Control.Set_True(Non_Empty);
   end Place_Item;

   procedure Extract_Item(Item: out Some_Type) is
      OK: Boolean;
   begin
      Buff.Extract(Item, OK);
      if not OK then
         Ada.Synchronous_Task_Control.Suspend_Until_True(Non_Empty);
         -- note this is a task activation event
         Buff.Extract(Item, OK); -- OK must be true
      end if;
      Ada.Synchronous_Task_Control.Set_True(Non_Full);
   end Extract_Item;

   protected body Buff is
      procedure Place(Item    : in Some_Type;
                      Success: out Boolean) is
      begin
         Success:= not Buffer_Full;
         if not Buffer_Full then
            -- put Item into Buffer
            Buffer_Empty:= False;
            -- set Buffer_Full if appropriate
         end if;
      end Place;

      procedure Extract(Item  : out Some_Type;
                        Success: out Boolean)) is
      begin
         Success:= not Buffer_Empty;
         if not Buffer_Empty then
            -- extract Item from Buffer
            Buffer_Full:= False;
            -- set Buffer_Empty if appropriate
         end if;
      end Extract;
   end Buff;
end Buffer;
```

7.12 Programming timeouts

There can be situations where a call to a protected object's entry should be retracted after a period of time if the event that should release it has not occurred. In full Ada, this would be:

```
select
   PO.Call;
   Timeout:= False;
or
   delay until Some_Time;
   Timeout:= True;
end select;
```

Identical functionality can be achieved in Ravenscar using an extra task that is event-triggered and a protected object that is used to pass the timeout value to this task. This is illustrated in the example below; it is important to note the expansion in code needed to accommodate this effect. The full language clearly has significant superior expressive power in this area as well as others.

EXAMPLE Programming timeouts in Ravenscar

```
protected PO is
   entry Call(Timeout: out Boolean);
   procedure Used_To_Release_Call;
   procedure Too_Late;
private
   Timed_Out: Boolean:= False;
   Release: Boolean:= False;
end PO;

protected body PO is
   procedure Too_Late is
   begin
      if Call'Count = 1 then
         Timed_Out:= True;
         Release:= True;
      end if;
   end Too_Late;
   procedure Used_To_Release_Call is
   begin
      Timed_Out:= False;
      Release:= True;
   end Used_To_Release_Call;
   entry Call(Timeout: out Boolean) when Release is
   begin
      Timeout:= Timed_Out;
      Release:= False;
      -- further non-suspending code if necessary
   end Call;
end PO;

protected Timer_Control is
   entry Wait(Wait_Time: out Ada.Real_Time.Time);
   procedure Set_Time(Wait_Time: Ada.Real_Time.Time);
private
   Timeout: Ada.Real_Time.Time;
   Released: Boolean:= False;
end Timer_Control;

protected body Timer_Control is
   entry Wait(Wait_Time: out Ada.Real_Time.Time) when Released is
   begin
      Wait_Time:= Timeout;
      Released:= False;
   end Wait;
   procedure Set_Time(Wait_Time: Ada.Real_Time.Time) is
   begin
      Timeout:= Wait_Time;
      Released:= True;
   end Set_Time;
end Timer_Control;
```

```
task Timer; -- note this task has more than one activation event

task body Timer is
   T: Ada.Real_Time.Time;
begin
   loop
      Timer_Control.Wait(T);
      delay until T;
      PO.Too_Late;
   end loop;
end Timer;

-- application calls the following
-- Timer_Control.Set_Time(Some_Time);
-- PO.Call(Timeout);
```

7.13 Further expansions to the expressive power of the Ravenscar profile

If static timing analysis is not of interest to the application program and a more general model of tasks and interrupts is required, this can still be achieved with reasonable expressive power within the subset definition. However, as noted earlier, the Ravenscar profile is not a substitute for the full language when that level of expressive power is needed.

Dynamic creation and termination of tasks can be simulated by declaring a pool of event-triggered tasks at program start-up, each containing an infinite loop which has a suspending operation as its first statement, such that its execution can be invoked dynamically by one of the task synchronization primitives. Thus, by changing the settings of suspension objects and entry barriers, it is possible for certain tasks to have their execution disabled while others have execution enabled.

Dynamic exchange of interrupt handlers, often required for applications performing mode change, can be simulated by embodying all the different handling code for a particular interrupt in one interrupt handler protected procedure, with each of the different actions being coded as case alternatives in a case statement, dependent on a mode selector. By changing the value of the mode selector, the same handler procedure can perform different response actions at various times during program execution.

Dynamic task priority change is also generally associated with mode change. This can be simulated using a separate event response task for each mode of operation (and assigning a different priority to each task as required), such that the execution of each task that belongs to a dormant mode is suspended until signalled when its mode becomes active.

A similar effect to requeue can be achieved by completing the protected entry body and returning a status result to the caller, which can then emit a subsequent protected entry call to the intended destination of the requeue statement. If each protected entry is called only by a single task, then this alternative technique does not introduce any race conditions.

Similarly, if static timing analysis is not of interest, the classic non-timed rendezvous operations can still be achieved within the subset definition by use of suspension objects for synchronization and protected object entries for communication.

No conditional form of suspension is supported by the Ravenscar profile. This can be simulated if a suspension object is used by polling the state of the suspension object (via the Current_State function in the package Ada.Synchronous_Task_Control), or if a protected entry is used by polling the value of the protected data which controls the synchronization (i.e. the barrier Boolean).

8 Verification of Ravenscar programs

8.1 General

The Introduction described the motivation for the Ravenscar profile in terms of the need to verify the temporal behaviour of concurrent real-time programs. This clause provides an introduction to the forms of verification that can applied to Ravenscar applications to deliver dependable systems.

The approach to verification in the presence of Ada tasking is similar in many ways to that traditionally used for cyclic executives. Each thread of control is independently verified for conformance with its precise/formal specification, for example by performing requirements-based testing or by use of static analysis tools on its sequential behaviour. Then, the program as a whole is verified against all its timing constraints. This latter stage differs from the cyclic executive approach in the presence of priority-based pre-emptive task scheduling, in that it can be automated by the use of, for example, a response time analysis (RTA) tool to verify that a given task set meets its deadlines. The tool-based approach greatly simplifies the process of verification of timing constraints during development, and of re-verification after the system has undergone modification during maintenance.

The effects of arbitrary dynamic pre-emption can be statically analysed by considering all accesses to the global state of the program as being volatile, e.g. two successive reads to the same global state variable can deliver different values (as for reads of values delivered by an external device).

The core set of Ravenscar profile run-time system packages can be developed to the most stringent software development standards so that these packages are suitable for inclusion in an application that requires certification against an applicable standard such as RTCA DO-178C.[3]

This clause looks at four levels of verification:

— static analysis of sequential code (8.2)

— static analysis of concurrent code (8.3)

— scheduling analysis (8.4)

— formal analysis (8.5)

8.2 Static analysis of sequential code

As discussed in the introduction, the Ravenscar profile is silent about those features of the sequential language that should be used with the profile (apart from requiring no implicit use of the heap). Similarly, it is not appropriate here to discuss the forms of static analysis that should be used to verify the functional behaviour of each task. See ISO/IEC TR 15942 for further information.

8.3 Static analysis of concurrent code

8.3.1 General

The two main goals of applying static analysis techniques to Ravenscar programs are:

— to obtain the same level of proof and data/information flow analysis for concurrent programs as is currently achievable for a sequential program;

— to obtain proof of absence of the concurrency-related run-time errors, to supplement the proof of absence of run-time errors that is currently achievable for sequential code.

The concurrency-related run-time errors that apply to Ravenscar programs are described in 6.3.4 and 6.3.5.

In addition, it is highly desirable to eliminate the implementation-defined effect of task termination in the presence of the No_Task_Termination restriction.

8.3.3 to 8.3.10 address various techniques for producing static analysis evidence to meet the above goals. These verification processes are made possible by the following assertions about the behaviour of a valid Ravenscar program:

— Each task and interrupt handler execution is deferred until after program elaboration is complete.

— Tasks do not terminate.

All task communication is made via protected shared variables preferably using protected objects, or atomic objects accompanied by a read/write protocol statically proven free of race conditions.

All protected shared variables are initialized during library-level elaboration code.

8.3.2 Program-wide information flow analysis

Current technology supports data flow analysis, information flow analysis, and proof based on pre- and post-conditions and invariants, for sequential code only. The goal is to extend this to Ravenscar programs that include tasks, protected objects and interrupt handlers.

The data dependency information that is currently used to analyse sequential programs can be applied to each task and each interrupt handler in the concurrent program as an independent entity. Thus, the existing tools and techniques can verify each thread of control in isolation, including its use of privately accessed global data. This then leaves only the issue of the verification of the interactions between the threads of control as represented by the set of protected shared variables.

The protected shared variables present in the program should be initialized by library-level elaboration code in order to ensure that no shared data are uninitialized at the point of use. If initialization was performed during the operation phase instead, a race condition can be introduced. For a suspension object, initialization is defined in ISO/IEC 8652 as occurring at the point of declaration. For a protected object or an atomic object, all fields should be initialized either as part of object elaboration, or using library-level package elaboration code. In conjunction with the use of pragma Partition_Elaboration_Policy(Sequential), this ensures that no thread of control can access any shared state that has not been fully initialized.

After the initialization phase is complete, the protected shared variables can be modelled for data and information flow analysis purposes, assuming that their data are volatile. Since the data can be updated at any time due to the effects of pre-emption and interrupt occurrence, any specific task necessarily assumes that the value of a protected shared variable can change at any time. For example, two successive reads by a task of a protected shared variable can deliver different results and similarly, the value read by a task following a write by the same task cannot be assumed to be the written value. This volatility is the same abstraction as that used to model access to external program data, such as that which has an address clause or is an imported variable (via the Import aspect). Thus, assuming that the static analysis technique supports access to volatile external data, concurrent access to protected data can be modelled in the same way. As a result, each thread of control can now be described both in terms of its sequential data and information flow, and in terms of its interactions with volatile protected shared variables.

Having obtained the analysis of each thread of control that includes its interactions with the protected state, it is then possible to combine the analyses to form the overall data and information flow for the program as a whole, across the task and interrupt handler boundaries. This allows the designer to make assertions about how the entire program should behave in terms of the effect that it has on its external inputs (including interrupts) to produce its external outputs. These assertions can then be verified by the analysis to the same degree of confidence as is currently achievable in a sequential program.

This form of static analysis does not address the timing or ordering properties of the program. 8.4.4 and 8.5 address these topics by describing the use of RTA and other forms of formal analysis, such as model checking, which can prove statically the timing properties of the program.

8.3.3 Absence of run-time errors

Existing static analysis techniques can be used to prove absence of run-time errors due to language-defined exceptions within sequential code. The corresponding guidance on the sequential code constructs that can be used to achieve this goal is contained in ISO/IEC TR 15942. These techniques can be independently applied to each individual thread of control (i.e. task, main program or interrupt handler) of a Ravenscar program.

In order to extend these existing techniques to a full Ravenscar program, the various forms of run-time check failure that relate directly to the concurrency features should be addressed. These can be broken down into the following groups:

— Errors during program elaboration, such as access-before-elaboration or use of uninitialized data.

— Errors after program elaboration is complete, during the normal operation phase of the application, in particular the exceptions that are cited in 6.3.4 and 6.3.5.

— Erroneous behaviour during normal operation, in particular concurrent access to unprotected shared variables (see 6.3.7).

— Implementation-defined behaviour as a result of violation of the No_Task_Termination restriction.

8.3.4 to 8.3.10 discuss various techniques that can be applied to verify statically that these forms of error cannot occur.

8.3.4 Elaboration errors

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. Having obtained a correct elaboration order during development, this ordering is usually predictable except when there is a switch to a different compiler vendor, or an upgrade to a new product version from the same vendor that uses a different algorithm for any units that have implementation-defined ordering. This implicit order variation can be prevented by the explicit use of elaboration order pragmas, once a correct order has been established.

Within a concurrent program however, access to global data that is not yet initialized by the elaboration code can occur as a result of race conditions that vary between development mode and deployment mode, due to factors such as the use of hardware of differing performance or memory access times, inclusion or exclusion of checking code, differences in interpretation of priority scheduling variations etc. These race conditions are more likely to be present because of the Ada rule that a library-level task is activated by its master package prior to the execution of that master's body elaboration code, and also prior to the execution of the elaboration code of later library units in the overall program elaboration order. Another contributing factor to the race condition is that having completed its activation, the Ada task proceeds into its normal execution code, and hence is programmed to immediately suspend to prevent this code from executing while program elaboration is still incomplete. Similar concerns apply to the execution of interrupt handlers after attachment. An interrupt can trigger execution of a handler prior to the completion of program elaboration, and in this case, the handler cannot be programmed to suspend. Such an error can actually occur silently. The task or interrupt handler can read an uninitialized value of a shared variable and not cause any exception to be raised, even in the presence of pragma Normalize_Scalars.

There are several solutions that can mitigate this hazard statically. The most obvious one is to ensure that all shared variables of a Ravenscar program are initialized at the point of declaration. However, this is inappropriate in the case when elaboration code in the body is needed to set a correct initial value. Logically, it is highly desirable to assert that the dynamic semantics of the program are unaffected whether global shared data are initialized at the point of declaration, or by library package body elaboration code, assuming a correct elaboration order for the sequential elaboration code has been enforced using elaboration control pragmas.

In order to achieve the static guarantee that all library units have been elaborated prior to the activation of any task and prior to the invocation of any interrupt handler, the Partition_Elaboration_Policy pragma was added to ISO/IEC 8652. If this pragma is used with argument Sequential, then all task activation and interrupt handler attachment is deferred until after all program elaboration code is complete, i.e. just prior to the call of the main subprogram (see also 6.3.8).

8.3.5 Execution errors causing exceptions

6.3.4 and 6.3.5 identify the concurrency-related run-time checks that are required of a conformant implementation of the Ravenscar profile. 8.3.6 to 8.3.10 examine techniques for static elimination of these error conditions.

8.3.6 Max_Entry_Queue_Length and suspension object check

The static detection of absence of entry queue length violation can be achieved by applying further constraints on the application code, namely that a maximum of one task object can call each protected entry.

This also implies that the task objects, protected objects and protected entries are statically identified. Static identification of an object excludes its name being determined dynamically such as via a function result, a dynamic array index, the dereferencing of an access value etc. More extensive analysis would be required for a less restrictive scheme showing that there is no program state in which more than one task can be calling the same protected object. Such analysis includes the use of model checking (see 4.5). The same approach can be applied to the static detection of absence of more than one task waiting on each suspension object at any time.

8.3.7 Priority ceiling violation check

The static detection of absence of priority ceiling violation can be achieved assuming the following further constraints:

— all task objects and protected objects have a static priority (this can be supplied via a static expression of a type discriminant for example);

— the protected object call chain (including nested protected object calls) that is made by each task object and each interrupt handler is statically determinable, by requiring static identification of the target protected object in all cases.

8.3.8 Potentially blocking operations in a protected action

The static detection of the absence of execution of a potentially blocking operation within a protected action is feasible, given the additional constraint on the use of indirect subprogram calls. This then allows the call trees to be statically determined. The presence of any of the following constructs in any protected or subprogram body in the call tree that is rooted in a protected operation body would then be statically disallowed:

— a protected entry_call_statement;

— a delay_statement;

— a call to Ada.Synchronous_Task_Control.Suspend_Until_True;

— a call to any other language-defined subprogram that is defined to be potentially blocking (see ISO/IEC 8652:2023, 9.5.1) .

In addition, the determination of the call trees would enable static detection of an external subprogram call with the same target protected object as that of the protected action, assuming the restriction that the target protected object is always statically identified.

A slightly less restrictive scheme is possible using formal verification methods such as model checking (see 4.5) to determine if a program state exists such that a protected action would cause execution of a potentially blocking operation (which may be within conditionally-executed code, although this style is not recommended).

It is also possible to support detection of potentially blocking operations in the presence of indirect procedure calls if a pre-condition that specifies a non-blocking property is asserted prior to each indirect call, and that property is shown to be satisfied statically by all possible procedures that can be invoked by that call. Similarly, the check for circularity in the protected object call chain is possible even in the case of non-statically identified protected objects, by imposing a pre-condition whereby none of the potentially called protected objects invoke operations of any protected objects that are higher in the call chain.

8.3.9 Task termination

The Ravenscar profile defines a static task set and prohibits dynamic task creation. The intent is that all tasks are created during program start-up, but in any mode of operation, some of them can be dormant, waiting on a synchronization event. A task that is no longer required to be executed would wait on its event indefinitely. In this model, task termination is considered to be an error case and hence the restriction No_Task_Termination is required by the Ravenscar profile. The effect of violation of the No_Task_Termination restriction is implementation-defined.

Task termination within the restrictions of the Ravenscar profile can occur only as a result of normal exit from the task body, or as a result of an unhandled exception.

The case of avoidance of normal exit can be statically analysed if a coding restriction is placed on the task body code. The final statement is either an infinite loop or else a compound statement (such as a conditional or case statement) that can only cause an infinite loop to be executed.

The case of showing absence of exceptions by static analysis has already been covered in 6.3.6 and 8.3.3 to 8.3.8.

The combination of these two techniques can be used to ensure statically that task termination cannot occur, and hence also that no implementation-defined behaviour that results from task termination can be invoked.

8.3.10 Use of unprotected shared variables

The intent of the Ravenscar profile is that tasks and interrupt handlers should not make concurrent use of an unprotected shared variable. All interactions involving tasks or interrupt handlers are recommended to be via protected or suspension objects, or by appropriate use of atomic objects (see 4.2.3). The avoidance of unprotected shared variables is generally a requirement of high integrity systems, although detection of this erroneous case is not mandated by the Ravenscar profile definition.

The static detection of the absence of unprotected shared variables can be achieved assuming the restriction that the use of all global variables of unprotected type by each task object and by each interrupt handler is statically identifiable. All visible objects of a protected type can be safely shared. Objects of an atomic type can be safely shared too, but only provided that they are accessed by a read/write sequence that is statically proven free of race conditions. In addition to that, static verification can then ensure that no unprotected global variable is accessed by more than one thread of control.

It should be noted that if a task object or interrupt handler shares global data only with program elaboration code, (i.e. the elaboration code initializes global data that is subsequently used privately by a single task or interrupt handler), then this data does not need protection if the Partition_Elaboration_Policy pragma is used with the argument Sequential. This is because the pragma ensures that the elaboration is complete prior to any task execution or interrupt attachment (and hence there can be no sharing violation).

8.4 Scheduling analysis

8.4.1 General

8.4 provides more details on the procedure to be followed. The aim is to introduce the form this analysis takes as it is not appropriate within this document to give a full tutorial on this material; such material can be found in text books (e.g. References [13] and [14]). Ravenscar facilitates the use of these techniques as it supports priority-based dispatching and ceiling locking on protected objects. To apply these techniques, however, further constraints on application code are recommended. All tasks should have a single invocation event and allow other parameters to be analysed or measured.

In 8.4.2 to 8.4.5, priority assignment is considered, then two forms of analysis are introduced: rate monotonic analysis and response time analysis.

8.4.2 Priority assignment

The use of priority-based pre-emptive dispatching defines a mechanism for scheduling. The scheduling policy is defined by the mapping of tasks to priority values. Many different schemes exist depending on the temporal characteristics of the task and other factors such as criticality. For hard deadline tasks, whose failure to meet a deadline can cause a failure of the application program, it is usually assumed that the following three parameters are known:

T – Period; time interval between consecutive arrivals of the task

D – Deadline; required latest completion time for the task (relative to its arrival)

C – Computation time; worst-case execution time needed for the task to complete one activation.

For periodic tasks, T is the time interval between releases. For sporadic tasks, T is the minimum inter-arrival time for the event that releases the task. The three parameters (T, D, C) are always given in the same time units. Therefore (30 ms, 20 ms, 2.73 ms) defines a task that (at maximum) is released every 30 ms; completes within 20 ms; and has a maximum computation time of 2.73 ms. These latter values are obtained either by measurement or by some form of static timing analysis (or a combination of the two).

If all tasks are hard and criticality itself is not taken into account (because all tasks are required to always meet their deadlines) then there is an optimal algorithm for assigning priority if D < = T for all tasks. The term “optimal” means that the algorithm is as good as any other fixed priority scheme. The optimal algorithm is called deadline monotonic and simply assigns priority based on deadline – the shorter the deadline the higher the priority. In the special case when D = T for all tasks, this scheme is known as rate monotonic.

An important property of fixed priority dispatching is that the lower priority tasks are the most vulnerable to missing a deadline if there is a run-time problem such as a task executing for more than its assumed maximum C. Because of this property, the systems designer can place the highly critical tasks at higher priorities than the deadline monotonic scheme would advise. This can reduce schedulability but is perfectly valid and amenable to response time analysis (see below).

Another reason to raise a task priority is to reduce jitter on input and/or output actions. Higher priority tasks have a more regular execution pattern and hence important events such as reading a sensor or writing to an actuator will occur with less variation from one period to the next. Scheduling analysis will only ensure that a task completes somewhere between its release and its deadline. One way of reducing jitter is thus to reduce the deadline of the tasks that perform jitter-sensitive I/O. If this is done, then the deadline monotonic priority assignment scheme will automatically allocate a higher priority.

Most scheduling schemes assume that each task is assigned a unique priority. Any Ada runtime for Ravenscar will support at least 32 priorities (and can indeed support many more). Although maximum schedulability does require distinct priorities for the tasks, it is unusual for an application to be so close to being unschedulable that it requires these unique priorities. Response time analysis can again deal with shared priority values. It should also be noted that that some real-time kernels can exploit the knowledge that tasks share priority to reduce the memory requirement. This is achieved by noting that two (or more) tasks that share a priority level never execute at the same time and hence can "share" a task stack.

Once a priority map has been agreed for the set of tasks within the application, the priorities for the protected objects can be assigned systematically.

8.4.3 Rate monotonic utilization-based analysis

For a constrained set of temporal characteristics, there exists a very simple schedulability test that quickly verifies if all deadlines will always be met. The constraints are that D = T for all tasks, and that priorities are assigned using the Rate Monotonic scheme. In practice this means that all tasks are hard and periodic. Each task is expected to finish before its next release and there is no additional requirement to control jitter. Assuming, initially, that the program does not contain protected objects (i.e. all tasks execute independently) the schedulability test is simply a matter of checking the utilization of the task set. For each task, the fraction of a complete processor required is given by C/T. If this is summed across all tasks, this gives the total utilization of the application. Clearly, this value cannot be more than 1.0 or the system will never be schedulable. The actual upper bound (which is less than 1.0) is given by the following formula, which is a function of n, the number of tasks in the system. See Formula (1):

$$\sum_{i:=1}^{n} \left(\frac{C_i}{T_i} \right) \leq n(2^{1/n} - 1) \qquad (1)$$

where

n is the number of tasks;

i is a running index across all individual tasks;

C_i is the worst-case computation time of task i;

T_i is the period of the task i.

As n gets arbitrarily large, this expression converges on a single value. This is the famous rate monotonic result, which states that a utilization of less than 0.69 will always furnish a schedulable system.

Once protected objects (POs) are introduced, blocking can occur. Here, when a task is released it can be prevented from executing by the currently 'low' priority task running with a 'high' ceiling value while in a PO. For each task, the maximum blocking time, B, can be calculated. This is the maximum time a lower priority task can execute with a priority equal to or higher than the task currently under consideration. As noted in 4.2.1, the use of a priority ceiling protocol such as Immediate Priority Ceiling Protocol (IPCP) on POs does reduce blocking to its minimum value. The utilization test is now augmented with the result that each task is examined in turn. Task j can be calculated using Formula (2):

$$\sum_{i:=1}^{j}\left(\frac{C_i}{T_i}\right)+\frac{B_j}{T_j}\leq j\left(2^{1/j}-1\right) \tag{2}$$

where B_i is the worst-case blocking time of task i.

The blocking term for the lowest-priority task is 0, as it cannot suffer blocking by definition.

The simplicity of the utilization-based test makes it a very attractive one to use. Yet, it only applies to the constrained set of task characteristics. Moreover, it is a sufficient but not necessary test. If the application passes the test, all timing constraints will be met. If it fails the test, instead, the system can still be schedulable. A better test is needed in these circumstances. 8.4.4 shows one such example.

8.4.4 Response time analysis

Response time analysis is a general technique, whose stipulations are both necessary and sufficient to the feasibility of a given task. Response time analysis will deal with any priority assignment scheme and any relationship, between D, the task deadline, and T, the task period, (although its simple form requires $D <= T$). and T, (although its simple form requires D < = T). Moreover, it is a necessary and sufficient scheme for most situations. Like the utilization-based method, it is easily incorporated into tools – many of which already exist.

The form of the analysis is quite straightforward. Firstly, the worst-case (longest) completion time for each task is calculated. This is known as the task response time, R. Secondly, these R values are compared, trivially, with the deadlines to make sure that R is less than D for all tasks. The response time equation is shown in Formula (3):

$$R_i = C_i + B_i + \sum_{j\in h(i)}\left\lceil\frac{R_i}{T_j}\right\rceil C_j \tag{3}$$

where R_i is the value to be computed by the fixed-point equation, which denotes the response time of task i.

As ceiling functions are used, the unit for time is chosen so that all parameters are represented as integers.

Formula (4) shows a solution by forming a recurrence relation.

$$\omega_i^{k+1} := C_i + B_i + \sum_{j \in h(i)} \left\lceil \frac{\omega_i^k}{T_j} \right\rceil C_j \quad (4)$$

where ω_i^{k+1} is the solution of the fixed point equation for task *i* at the *k*-th iteration of the calculation.

The initial value of the iteration variable is the task's computation time. Iteration continues until either the same value is obtained on two successive iterations (in which case the response time has been calculated) or the value rises higher than the task's deadline (in which case the task is not schedulable).

The above description represents the textbook version of the analysis. The engineering version requires extra terms to capture the overhead of actual implementation. Firstly, overheads such as context switches can be assigned to the task that caused them (by incorporating them into the C parameter). Next, the kernel overheads associated with manipulating the delay queue, handling clock interrupts and the releasing of tasks, is factored in. The specific form this takes will depend on the structure of the kernel, but the kernel necessarily provides the data needed to model this overhead. This is a documentation requirement specified in ISO/IEC 8652:2023, Annex D, which is discussed further in 8.4.5.

NOTE For an example on how to include this term in the analysis, see References [13] and [14].

Finally, the overheads incurred by the application's interrupts are accounted for. A known bound on the arrival of such interrupts, and the execution time of each attached handler, is important. Putting these values together allows a set of interrupt overhead terms to be included in the response time analysis.

The appropriate use of the Ravenscar profile and the scheduling results outlined in 8.4.1 to 8.4.3 provide a sound engineering basis for constructing high integrity real-time systems. The theory is mature and tool support is available.

8.4.5 Documentation requirement on run-time overhead parameters

There are a number of places in ISO/IEC 8652:2023 where documentation requirements and metrics are required of an implementation. Those of most relevance to Ravenscar are:

— ISO/IEC 8652:2023, C.3 concerning the interrupt model;

— ISO/IEC 8652:2023, D.2.3 concerning maximum duration of priority inversion;

— ISO/IEC 8652:2023, D.8 concerning clock accuracy;

— ISO/IEC 8652:2023, D.9 concerning the precision of delay until;

— ISO/IEC 8652:2023, D.12 concerning interrupt blocking;

— ISO/IEC 8652:2023, D.12 concerning overhead involved with the use of protected objects.

Unfortunately, this is not a comprehensive list of the data needed to fully model the overheads caused by the run-time system. Typically, it is also important to have:

— the cost of context switches between tasks;

— the cost of handling delay queue operations.

Both of these factors can, contingent on the implementation of queues with the run-time system, depend on the number of tasks in the application's program. Nevertheless, if timing analysis is used on a Ravenscar program, it is recommended to have one of the following:

— evidence of all necessary parameters;

— a means by which the programmer can measure these parameters;

— formulae by which these parameters can be calculated.

8.5 Formal analysis of Ravenscar programs

The Ravenscar profile supports only a simple concurrency model with the error conditions being relatively easy to avoid. For example, the use of shared resources (via projected objects with ceiling priorities) cannot lead to deadlock. Nevertheless, to gain a very high level of assurance, it can be necessary to formally analyse a Ravenscar program. As outlined in 4.5, such analysis takes the form of either mechanized proof (via a theorem prover) or model checking.

There is already experience of using model checking to validate Ravenscar programs. It is possible to add worst-case and best-case execution times for state transitions and to then check that deadlines are never missed. Alternatively, model checking can be used to validate the top-level description of the timing constraints, leaving scheduling analysis to check deadline satisfaction once execution times from the implementation are known. This type of verification method can check that the end-to-end deadlines of some tasks are met provided that each individual tasks meets its own timing requirements. Each task is represented by an automaton and each protected object by a shared variable.

NOTE There are no problems with mutual exclusion in these formal models.

As with Ada itself, there can never be a formal map between a Ravenscar program and its model. However, the use of standard paradigms and libraries of associated (reusable) models allows a high integrity process to be defined.

This demonstrates that formal approach can be applied effectively to Ravenscar programs, but this does not imply that all high integrity Ravenscar programs need this level of verification. For many systems, static analysis of each task is sufficient to generate the appropriate level of confidence.

9 Extended example

9.1 General

The example presented in this clause is designed to illustrate the expressive power of the Ravenscar profile and the associated coding paradigms (discussed in Clause 7), which aim to facilitate off-line scheduling analysis in the form outlined in 8.4.

The extended application example uses all of the concurrency components permitted by the Ravenscar profile. The structure of the example models the operation of real-world embedded real-time systems on a reduced and simplified scale. The presentation of the example also outlines the information required for, and obtained from, the execution of deadline monotonic priority assignment and off-line scheduling analysis.

9.2 Ravenscar application example

The example system includes a periodic process that handles orders for a variable amount of workload. Whenever the request level exceeds a certain threshold, the periodic process farms the excess load out to a supporting sporadic process. While such orders are executed, the system can receive interrupt requests from an external source. Each interrupt treatment records an entry in an activation log. When specific conditions hold, the periodic process releases a further sporadic process to perform a check on the interrupt activation entries recorded in the intervening period. The policy of work delegation adopted by the system allows the periodic process to ensure the constant discharge of a guaranteed level of workload. The correct implementation of this policy also requires assigning the periodic process a higher priority than those assigned to the sporadic processes, so that guaranteed work can be performed in preference to subsidiary activities.

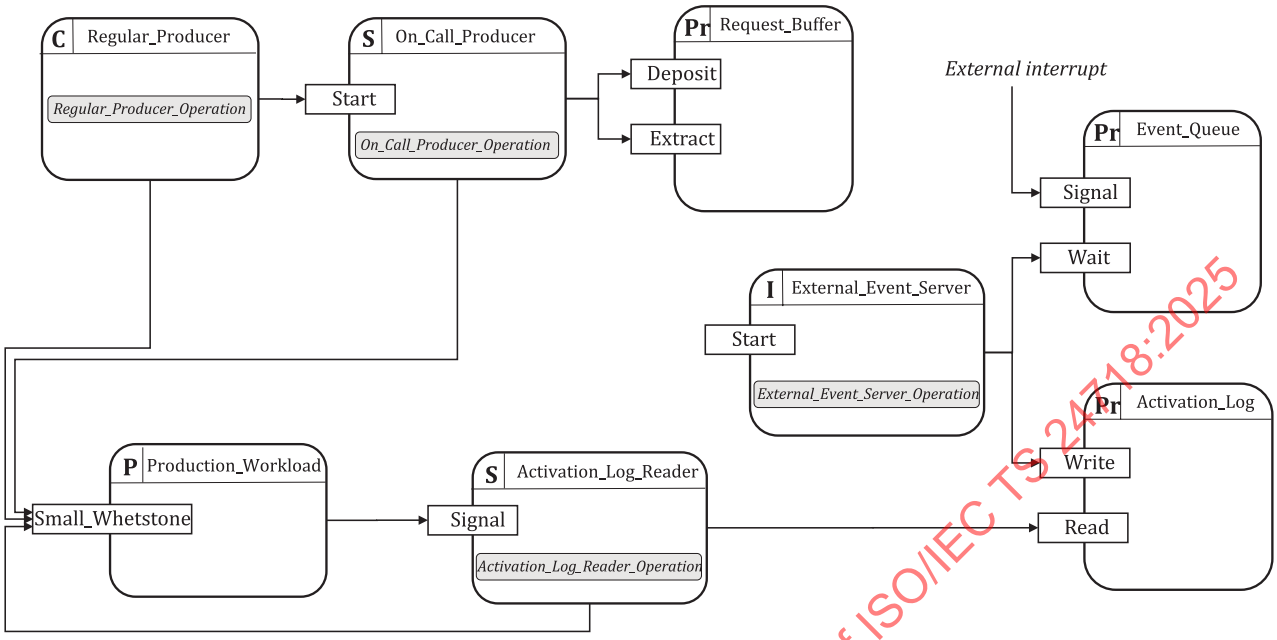
Figure 1 shows an HRT-HOOD-like representation of the system,[15] while the legend recalls the meaning of the symbols and notations used in the diagram.

In HRT-HOOD terms, the system comprises:

— 4 active (i.e. threaded) objects respectively called Regular_Producer, On_Call_Producer, Activation_Log_Reader and External_Event_Server;

— 1 passive (i.e. unthreaded) object called Production_Workload;

— 3 protected objects respectively called Request_Buffer, Event_Queue and Activation_Log.



Key

C	cyclic object (threaded object incorporating a periodic task)
S	sporadic object (threaded object incorporating a sporadic task)
Pr	protected object (unthreaded object providing protected operations)
I	interrupt sporadic object (threaded object incorporating sporadic task with release event delivered by an interrupt)
P	passive object (unthreaded object providing unprotected operations)
AAA	operation exported by the object for users to invoke
→	object invocation of exported operation
AAA	internal operation of threaded object

Figure 1 — Schematic architecture of the example Ravenscar application

The operation of the system proceeds as follows:

Regular_Producer, which Figure 1 tags as Cyclic, embeds a fixed-rate periodic task that carries out a given amount of workload. The example represents the execution of this workload by the invocation of the well-known Small_Whetstone procedure exported by the shared Passive object Production_Workload.

When Regular_Producer determines that the required amount of workload exceeds its ceiling capacity, it delegates the excess workload out to On_Call_Producer. On_Call_Producer, which Figure 1 tags as Sporadic, embeds a sporadic task whose activation is specifically invoked to take over the excess workload of Regular_Producer.

The sporadic activation and the associated workload transfer occur by means of a typical Ravenscar data-oriented synchronization: Regular_Producer invokes the Start operation exported by On_Call_Producer with a parameter characterizing the service request. The Start operation enqueues the request in a private queue embedded within the Protected object Request_Buffer. The buffer is protected because new service requests are allowed to come in while the sporadic task is busy executing old ones. This follows from the decision to assign Regular_Producer a higher base priority than that of On_Call_Producer, which ensures the discharge of a guaranteed level of workload in preference to the execution of subsidiary activities.

A successful enqueueing releases the On_Call_Producer sporadic task, which indefinitely waits on an empty queue. The sporadic task fetches the request parameter from the top of the queue and performs the requested amount of workload in the same way as Regular_Producer. An invocation of Start fails when the queue held within Request_Buffer is full; for example, as a result of a (transient) rate of requests faster than service execution. Static analysis of the relationship between the maximum frequency of activation requests and the longest service time incurred by the sporadic task of On_Call_Producer should be used to prevent failure events of this kind.

While the system carries out the required level of workload (whether regular or excess), an external device can occasionally raise an interrupt to signal its call for attention. In keeping with the Ravenscar programming model, the example application maps the arrival of the external interrupt to the invocation of a protected procedure. Object Event_Queue exports the procedure in question, called Signal.

The service associated with the raising of the interrupt is carried out by the sporadic task embedded in External_Event_Server, which is tagged Interrupt-activated sporadic. To simplify the coding of the example, and in keeping with the programming model that minimizes the amount of activity performed at interrupt priority, the extent of this interrupt service is limited to the storing of an activation record in a protected buffer. The recording occurs by invocation of procedure Write exported by Protected object Activation_Log. The use of a protected buffer to hold the activation record offers the natural mechanism to preserve data integrity in the face of independent read and write activities.

In order for the system to monitor the arrival of service requests from the external device, when certain conditions hold, the periodic process embedded in Regular_Producer requests the task embedded in the Sporadic object Activation_Log_Reader to examine the latest activation record stored by the interrupt service carried out by External_Event_Server. Activation_Log_Reader does this by invoking the Read procedure of Activation_Log. This style of work partitioning between Regular_Producer and Activation_Log_Reader uses the Ravenscar concurrency mechanisms to allocate activities with differing degrees of importance to distinct tasks. This approach aids system modelling. It also favours the specialization of tasks, which is a way of using the Ravenscar profile definition to facilitate static analysis of the system.

The activation request issued by Regular_Producer for this purpose uses the other form of synchronization permitted by the Ravenscar profile: the data-less synchronization supported by suspension objects. Procedure Signal exported by Activation_Log_Reader performs this synchronization on a suspension object internally held by the object. As HRT-HOOD provides no specific object representation for suspension objects, the adopted convention is that procedures by the name Signal exported by Sporadic objects are understood as implemented by invocation of a private suspension object embedded within the object. Conversely, procedures by the name Start exported by Sporadic objects are implemented by invocation of the Deposit procedure exported by an associated Protected object.

NOTE Signal is also the name of the protected procedure attached to an interrupt, which dispatches the activation event to Interrupt-activated sporadic objects.

9.3 Code

9.3.1 General

The Ravenscar profile model does not inherently require the application to use any particular coding style for the execution of cyclic and sporadic tasks, protected objects, and interrupt handlers. However, if the application is required to pass schedulability analysis, certain task templates (patterns or stereotypes) and corresponding coding styles are useful in defining the activities that are to be analysed. These task templates are described in Clause 7 and are used to code the example application outlined in 9.2.

In order to emphasize the stereotypical nature of the task templates in the example, parametric components of the application code have been relegated into support packages named with "_Parameters" trailer added to the name of the corresponding base package.

NOTE The code of these support packages is provided in 9.3.4 to 9.3.6.

The Ravenscar-compliant HRT-HOOD coding convention has individual terminal objects in the system implemented as distinct library-level packages that carry the name of the corresponding object. An HRT-