Guidelines for Safe and Secure Ada/SPARK

Patrick Rogers
and Michael Frank

LEARN.
ADACORE.COM

LEARN.
ADACORE.COM

Guidelines for Safe and Secure Ada / SPARK

Release 2024-06

Patrick Rogers
and Michael Frank

Jun 30, 2024

CONTENTS

This document provides a reasonable set of coding standards to be applied to Ada/SPARK source code. The contents can be used as-is, or customized for a particular project.

This document was originally written by Patrick Rogers, and modified by Michael Frank.

[1] http://creativecommons.org/licenses/by-sa/4.0

CHAPTER

ONE

INTRODUCTION

Ada is a general purpose, high-level programming language designed to support the construction of long-lived, highly-reliable applications. Like all general-purpose languages, only a subset of the full language is appropriate for safety-critical applications because the full language includes facilities that are difficult to analyze and verify to the degree required. This document facilitates identification of subsets appropriate for the highest levels of integrity, including safety-critical applications.

SPARK is a statically verifiable subset of Ada designed specifically for the most critical applications. Ada constructs not amenable to verification are precluded, such as arbitrary use of access types and full tasking. SPARK is also a superset of Ada, with additional contracts for specifying and verifying programs. Many of the guidelines (and more) are implicit in the design of SPARK.

Therefore, this document defines guidelines for the development of high-integrity, safety-critical applications in either the Ada or SPARK programming languages, or both (because the two can be mixed).

1.1 Scope

This document provides guidelines for development decisions, both at the system level and at the unit level, regarding the use of the programming languages Ada and SPARK, as well as related tools, such as static analyzers and unit test generators. It is not concerned with presentation issues such as naming, use of whitespace, or the like.

1.2 Structure

Rather than defining a specific set of rules defining a single subset, this document defines a set of criteria, in the form of guidelines, used by system architects to identify project-specific subsets appropriate to a given project.

The guidelines are separated into related categories, such as storage management, object-oriented programming, concurrency management, and so on. Each guideline is in a separate table, specifying the rule name, a unique identifier, and additional attributes common to each table.

1.3 Enforcement

Detection and enforcement mechanisms are indicated for each guideline. These mechanisms typically consist of the application of a language standard pragma named `Restrictions`, with policy-specific restriction identifiers given as parameters to the pragma [AdaRM2016]. Violations of the given restrictions are then detected and enforced by the Ada compiler.

Alternatively, the AdaCore GNATcheck utility program has rules precisely corresponding to those restriction identifiers, with the same degree of detection and enforcement. For example, the language restriction identifier `No_Unchecked_Deallocation` corresponds to the GNATcheck **+RRestrictions:No_Unchecked_Deallocation** rule.

The advantage of GNATcheck over the compiler is that all generated messages will be collected in the GNATcheck report that can be used as evidence of the level of adherence to the coding standard. In addition, GNATcheck provides a mechanism to deal with accepted exemptions.

In some cases the enforcement mechanism is the SPARK language and analyzer. Many of the guidelines (and more) are implicit in the design of SPARK and are, therefore, automatically enforced.

In some (very) rare cases the enforcement mechanism is manual program inspection, although alternatives (e.g., SPARK) are usually available and recommended. These guidelines are included because they are considered invaluable in this domain.

1.4 About the Rules

Although we refer to them as **rules** in the tables for the sake of brevity, these entries should be considered **guidance** because they require both thought and consideration of project-specific characteristics. For example, in some cases the guidance is to make a selection from among a set of distinct enumerated policies. In other cases a single guideline should be followed but not without some exceptional situations allowing it to be violated. The project lead should consider which guidelines to apply and how best to apply each guideline selected.

Many of these rules can also be considered *good* programming practices. As such, many of them can be directly correlated to the *ISO/IEC Guidance to Avoiding Vulnerabilities in Programming Languages* [TR24772]. When a rule addresses one of these vulnerabilities, it is listed in the appropriate subsection.

CHAPTER

TWO

DEFINITIONS

This section contains terms and values used in the definitions of the rules set forth in this chapter.

2.1 Level

Level is the compliance level for the rule. Possible values are:

Mandatory
: Non-compliance with a *Mandatory* recommendation level corresponds to a **high risk** of a software bug. There would need to be a good reason for non-conformity to a mandatory rule and, although it is accepted that exceptional cases may exist, any non-conformance should be accompanied by a clear technical explanation of the exceptional circumstance.

Required
: Non-compliance with a *Required* recommendation level corresponds to a **medium to high risk** of a software bug. Much like a *Mandatory* recommendation, there would need to be a good reason for non-conformity to a required rule. Although it is accepted that more exceptional cases may exist, non-conformance should be accompanied by a clear technical explanation of the exceptional circumstance.

Advisory
: Failure to follow an *Advisory* recommendation does not necessarily result in a software bug; the risk of a direct correlation between non-conformance of an advisory rule and a software bug is low. Non-compliance with an advisory recommendation level does not require a supporting technical explanation, however, as the quality of the code may be impacted, the reason for the non-conformance should be understood.

2.2 Remediation

Remediation indicates the the level of difficulty to modify/update code that does not follow this particular rule.

High
: Failure to follow this rule will likely cause an unreasonable amount of modifications/updates to bring the code base into compliance.

Medium
: Failure to follow this rule will likely cause a large amount of modifications/updates to bring the code base into compliance, but those changes may still be cost-effective.

Low
Failure to follow this rule may cause a small amount of modifications/updates to bring the code base into compliance, but those changes will be minor compared to the benefit.

N/A
This rule is more of a design decision (as opposed to a coding flaw) and therefore, if the rule is violated, it is done so with a specific purpose.

CHAPTER

THREE

DYNAMIC STORAGE MANAGEMENT (DYN)

Goal

Maintainability
✓

Reliability
✓

Portability
Performance
✓

Security
✓

Description
Have a plan for managing dynamic memory allocation and deallocation.

Rules
DYN01, DYN02, DYN03, DYN04, DYN05, DYN06

In Ada, objects are created by being either *declared* or *allocated*. Declared objects may be informally thought of as being created "on the stack" although such details are not specified by the language. *Allocated* objects may be thought of as being allocated "from the heap", which is, again, an informal term. Allocated objects are created by the evaluation of allocators represented by the reserved word `new` and, unlike declared objects, have lifetimes independent of scope.

The terms *static* and *dynamic* tend to be used in place of *declared* and *allocated*, although in traditional storage management terminology all storage allocation in Ada is dynamic. In the following discussion, the term *dynamic allocation* refers to storage that is allocated by allocators. *Static* object allocation refers to objects that are declared. *Deallocation* refers to the reclamation of allocated storage.

Unmanaged dynamic storage allocation and deallocation can lead to storage exhaustion; the required analysis is difficult under those circumstances. Furthermore, access values can establish aliases that complicate verification, and explicit deallocation of dynamic storage can lead to specific errors (e.g., "double free", "use after free") having unpredictable results. As a result, the prevalent approach to storage management in high-integrity systems is to disallow dynamic management techniques completely. [SEI-C] [MISRA2013] [Holzmann2006] [ISO2000]

However, restricted forms of storage management and associated feature usage can support the necessary reliability and analyzability characteristics while retaining sufficient expressive power to justify the analysis expense. The following sections present possible approaches, including the traditional approach in which no dynamic behavior is allowed. Individual projects may then choose which storage management approach best fits their requirements and apply appropriate tailoring, if necessary, to the specific guidelines.

Realization
There is a spectrum of management schemes possible, trading ease of analysis against

increasing expressive power. At one end there is no dynamic memory allocation (and hence, deallocation) allowed, making analysis trivial. At the other end, nearly the full expressive power of the Ada facility is available, but with analyzability partially retained. In the latter, however, the user must create the allocators in such a manner as to ensure proper behavior.

Rule DYN01 is Required, as it avoids problematic features whatever the strategy chosen. Rules DYN02-05 are marked as Advisory, because one of them should be chosen and enforced throughout a given software project.

3.1 Common High Integrity Restrictions (DYN01)

Level → Required

Category

Safety
✓

Cyber
✓

Goal

Maintainability
✓

Reliability
✓

Portability
Performance
Security
✓

Remediation → Low

Verification Method → Compiler restrictions

3.1.1 Reference

Ada Reference Manual: H.4 High Integrity Restrictions[2]

3.1.2 Description

The following restrictions must be in effect:

- `No_Anonymous_Allocators`
- `No_Coextensions`
- `No_Access_Parameter_Allocators`
- `Immediate_Reclamation`

The first three restrictions prevent problematic usage that, for example, may cause unreclaimed (and unreclaimable) storage. The last restriction ensures any storage allocated by the compiler at run-time for representing objects is reclaimed at once. (That restriction does not apply to objects created by allocators in the application.)

[2] http://www.ada-auth.org/standards/12rm/html/RM-H-4.html

3.1.3 Applicable Vulnerability within ISO TR 24772-2

- 4.10 Storage Pool

3.1.4 Noncompliant Code Example

For No_Anonymous_Allocators:

```
X : access String := new String'("Hello");
...
X := new String'("Hello");
```

For No_Coextensions:

```
type Object (Msg : access String) is ...
Obj : Object (Msg => new String'("Hello"));
```

For No_Access_Parameter_Allocators:

```
procedure P (Formal : access String);
...
P (Formal => new String'("Hello"));
```

3.1.5 Compliant Code Example

For No_Anonymous_Allocators, use a named access type:

```
type String_Reference is access all String;
S : constant String_Reference := new String'("Hello");
X : access String := S;
...
X := S;
```

For No_Coextensions, use a variable of a named access type:

```
type Object (Msg : access String) is ...
type String_Reference is access all String;
S : String_Reference := new String'("Hello");
Obj : Object (Msg => S);
```

For No_Access_Parameter_Allocators, use a variable of a named access type:

```
procedure P (Formal : access String);
type String_Reference is access all String;
S : String_Reference := new String'("Hello");
...
P (Formal => S);
```

3.1.6 Notes

The compiler will detect violations of the first three restrictions. Note that GNATcheck can detect violations in addition to the compiler.

The fourth restriction is a directive for implementation behavior, not subject to source-based violation detection.

3.2 Traditional Static Allocation Policy (DYN02)

Level → Advisory

Category

Safety
✓

Cyber
✓

Goal

Maintainability
✓

Reliability
✓

Portability
Performance
Security
✓

Remediation → Low

Verification Method → Compiler restrictions

3.2.1 Reference

MISRA C Dir 4.12: "Dynamic memory allocation shall not be used."

3.2.2 Description

The following restrictions must be in effect:

- `No_Allocators`
- `No_Task_Allocators`

Under the traditional approach, no dynamic allocations and no deallocations occur. Only declared objects are used and no access types of any kind appear in the code.

Without allocations there is no issue with deallocation as there would be nothing to deallocate. *Heap* storage exhaustion and fragmentation are clearly prevented although storage may still be exhausted due to insufficient stack size allotments.

In this approach the following constructs are not allowed:

- Allocators
- Access-to-constant access types

- Access-to-variable access types
- User-defined storage pools
- Unchecked Deallocations

3.2.3 Applicable Vulnerability within ISO TR 24772-2

- 4.10 Storage Pool

3.2.4 Noncompliant Code Example

Any code using the constructs listed above.

3.2.5 Compliant Code Example

N/A

3.2.6 Notes

The compiler, and/or GNATcheck, will detect violations of the restrictions.

3.3 Access Types Without Allocators Policy (DYN03)

Level → Advisory

Category

Safety
✓

Cyber
✓

Goal

Maintainability
✓

Reliability
✓

Portability
Performance
Security
✓

Remediation → Low

Verification Method → Compiler restrictions

3.3.1 Reference

MISRA C Rule 21.3: "The memory allocation and deallocation functions of <stdlib.h> shall not be used."

3.3.2 Description

The following restrictions must be in effect:

- `No_Allocators`
- `No_Dependence => Ada.Unchecked_Deallocation`

In this approach dynamic access values are only created via the attribute `'Access` applied to aliased objects. Allocation and deallocation never occur. As a result, storage exhaustion cannot occur because no *dynamic* allocations occur. Fragmentation cannot occur because there are no deallocations.

In this approach the following constructs are not allowed:

- Allocators
- User-defined storage pools
- Unchecked Deallocations

Aspects should be applied to all access types in this approach, specifying a value of zero for the storage size. Although the restriction `No_Allocators` is present, such clauses may be necessary to prevent any default storage pools from being allocated for the access types, even though the pools would never be used. A direct way to accomplish this is to use **`pragma`** *`Default_Storage_Pool`* with a parameter of **`null`** like so:

```
pragma Default_Storage_Pool (null);
```

The above would also ensure no allocations can occur with access types that have the default pool as their associated storage pool (per Ada Reference Manual: 13.11.3 (6.1/3) Default Storage Pools[3]).

3.3.3 Applicable Vulnerability within ISO TR 24772-2

- 6.14 Dangling reference to heap [XYK]

3.3.4 Noncompliant Code Example

Any code using the constructs listed above.

[3] http://www.ada-auth.org/standards/12rm/html/RM-13-11-3.html

3.3.5 Compliant Code Example

```
type Descriptor is ...;
type Descriptor_Ref is access all Descriptor;
...
Device : aliased Descriptor;
...
P : Descriptor_Ref := Device'Access;
...
```

3.3.6 Notes

The compiler, and/or GNATcheck, will detect violations of the restrictions.

3.4 Minimal Dynamic Allocation Policy (DYN04)

Level → Advisory

Category

Safety
✓

Cyber
✓

Goal

Maintainability
✓

Reliability
✓

Portability
Performance
Security

Remediation → Low

Verification Method → Compiler restrictions

3.4.1 Reference

Power of Ten rule 3: "Do not use dynamic memory allocation after initialization."

3.4.2 Description

The following restrictions must be in effect:

- `No_Local_Allocators`
- `No_Dependence => Ada.Unchecked_Deallocation`

In this approach dynamic allocation is only allowed during "start-up" and no later. Deallocations never occur. As a result, storage exhaustion should never occur assuming the initial allotment is sufficient. This assumption is as strong as when using only declared objects on the "stack" because in that case a sufficient initial storage allotment for the stack must be made.

In this approach the following constructs are not allowed:

- Unchecked Deallocations

Note that some operating systems intended for this domain directly support this policy.

3.4.3 Applicable Vulnerability within ISO TR 24772-2

- 4.10 Storage Pool

3.4.4 Noncompliant Code Example

Any code using the constructs listed above.

3.4.5 Compliant Code Example

Code performing dynamic allocations any time prior to an arbitrary point designated as the end of the "startup" interval.

3.4.6 Notes

The compiler, and/or GNATcheck, will detect violations of the restrictions.

3.5 User-Defined Storage Pools Policy (DYN05)

Level → Advisory

Category

Safety
✓

Cyber
✓

Goal

Maintainability
✓

Reliability
✓

Portability	
Performance	
Security	✓

Remediation → Low

Verification Method → Code inspection

3.5.1 Reference

MISRA C Rule 21.3: "The memory allocation and deallocation functions of <stdlib.h> shall not be used."

3.5.2 Description

There are two issues that make storage utilization analysis difficult:

1. the predictability of the allocation and deallocation implementation, and
2. how access values are used by the application.

The behavior of the underlying implementation is largely undefined and may, for example, consist of calls to the operating system (if present). Application code can manipulate access values beyond the scope of analysis.

Under this policy, the full expressive power of access-to-object types is provided but one of the two areas of analysis difficulty is removed. Specifically, predictability of the allocation and deallocation implementation is achieved via user-defined storage pools. With these storage pools, the implementation of allocation (`new`) and deallocation (instances of `Ada.Unchecked_Deallocation`) is defined by the pool type.

If the pool type is implemented with fixed-size blocks on the stack, allocation and deallocation timing behavior are predictable.

Such an implementation would also be free from fragmentation.

Given an analysis providing the worst-case allocations and deallocations, it would be possible to verify that pool exhaustion does not occur. However, as mentioned such analysis can be quite difficult. A mitigation would be the use of the "owning" access-to-object types provided by SPARK.

In this approach no storage-related constructs are disallowed unless the SPARK subset is applied.

3.5.3 Applicable Vulnerability within ISO TR 24772-2

- 4.10 Storage Pool

3.5.4 Noncompliant Code Example

Allocation via an access type not tied to a user-defined storage pool.

3.5.5 Compliant Code Example

```
Heap : Sequential_Fixed_Blocks.Storage_Pool
         (Storage_Size => Required_Storage_Size,
          Element_Size => Representable_Obj_Size,
          Alignment    => Representation_Alignment);
type Pointer is access all Unsigned_Longword with
   Storage_Pool => Heap;
Ptr : Pointer;
...
Ptr := new Unsigned_Longword; -- from Heap
```

3.5.6 Notes

Enforcement of this approach can only be provided by manual code review unless SPARK is used.

However, the User-Defined Storage Pools Policy can be enforced statically by specifying `Default_Storage_Pool (null)`. This essentially requires all access types to have a specified storage pool if any allocators are used with the access type.

3.6 Statically Determine Maximum Stack Requirements (DYN06)

Level → Required

Category

Safety
✓

Cyber
✓

Goal

Maintainability
✓

Reliability
✓

Portability
Performance
Security

Remediation → Low

Verification Method → Static analysis tools

3.6.1 Reference

N/A

3.6.2 Description

Each Ada application task has a stack, as does the "environment task" that elaborates library packages and calls the main subprogram. A tool to statically determine the maximum storage required for these stacks must be used, per task.

This guideline concerns another kind of dynamic memory utilization. The previous guidelines concerned the management of storage commonly referred to as the "heap." This guideline concerns the storage commonly referred to as the "stack." (Neither term is defined by the language, but both are commonly recognized and are artifacts of the underlying run-time library or operating system implementation.)

3.6.3 Applicable Vulnerability within ISO TR 24772-2

- 4.10 Storage Pool

3.6.4 Noncompliant Code Example

N/A

3.6.5 Compliant Code Example

N/A

3.6.6 Notes

The GNATstack[4] tool can statically determine the maximum requirements per task.

[4] http://docs.adacore.com/live/wave/gnatstack/html/gnatstack_ug/index.html

CHAPTER

FOUR

SAFE RECLAMATION (RCL)

Goal

Maintainability
✓

Reliability
✓

Portability
Performance
✓

Security
✓

Description
Related to managing dynamic storage at the system (policy) level, these statement-level rules concern the safe reclamation of access (*pointer*) values.

Rules
RCL01, RCL02, RCL03

4.1 No Multiple Reclamations (RCL01)

Level → Mandatory

Category

Safety
✓

Cyber
✓

Goal

Maintainability
✓

Reliability
✓

Portability
✓

Performance
Security
✓

Remediation → High

Verification Method → Code inspection

4.1.1 Reference

[CWE2019] CWE-415: Double Free

4.1.2 Description

Never deallocate the storage designated by a given access value more than once.

4.1.3 Applicable Vulnerability within ISO TR 24772-2

- 6.39 Memory leak and heap fragmentation [XYL]

4.1.4 Noncompliant Code Example

```
   type String_Reference is access all String;
   procedure Free is new Ada.Unchecked_Deallocation
      (Object => String, Name => String_Reference);
   S : String_Reference := new String'("Hello");
   Y : String_Reference;
begin
   Y := S;
   Free (S);
   Free (Y);
```

4.1.5 Compliant Code Example

Remove the call to `Free (Y)`.

4.1.6 Notes

Enforcement of this rule can be provided by manual code review, unless deallocation is forbidden via `No_Unchecked_Deallocation` or SPARK is used, as ownership analysis in SPARK detects such cases. Note that storage utilization analysis tools such as Valgrind can usually find this sort of error. In addition, a GNAT-defined storage pool is available to help debug such errors.

4.2 Only Reclaim Allocated Storage (RCL02)

Level → Mandatory

Category

Safety
✓

Cyber
✓

Goal

Maintainability
✓

Reliability
✓

Portability
✓

Performance
Security
✓

Remediation → High

Verification Method → Code inspection

4.2.1 Reference

[SEI-C] MEM34-C: Only Free Memory Allocated Dynamically

4.2.2 Description

Only deallocate storage that was dynamically allocated by the evaluation of an allocator (i.e., `new`).

This is possible because Ada allows creation of access values designating declared (aliased) objects.

4.2.3 Applicable Vulnerability within ISO TR 24772-2

- 6.39 Memory leak and heap fragmentation [XYL]

4.2.4 Noncompliant Code Example

```
   type String_Reference is access all String;
   procedure Free is new Ada.Unchecked_Deallocation
     (Object => String, Name => String_Reference);
   S : aliased String := "Hello";
   Y : String_Reference := S'Access;
begin
   Free (Y);
```

4.2.5 Compliant Code Example

Remove the call to `Free (Y)`.

4.2.6 Notes

Enforcement of this rule can only be provided by manual code review, unless deallocation is forbidden via `No_Unchecked_Deallocation`.

4.3 Only Reclaim to the Same Pool (RCL03)

Level → Mandatory

Category

Safety
✓

Cyber
✓

Goal

Maintainability
✓

Reliability
✓

Portability
✓

Performance
Security
✓

Remediation → High

Verification Method → Code inspection

4.3.1 Reference

N/A

4.3.2 Description

When deallocating, ensure that the pool to which the storage will be returned is the same pool from which it was allocated. Execution is erroneous otherwise, meaning anything can happen (Ada Reference Manual: 13.11.2 (16) Unchecked Storage Deallocation[5]).

Each access type has an associated storage pool, either implicitly by default, or explicitly with a storage pool specified by the programmer. The implicit default pool might not be the same pool used for another access type, even an access type designating the same subtype.

[5] http://www.ada-auth.org/standards/12rm/html/RM-13-11-2.html

4.3.3 Applicable Vulnerability within ISO TR 24772-2

- 6.39 Memory leak and heap fragmentation [XYL]

4.3.4 Noncompliant Code Example

```
   type Pointer1 is access all Integer;
   type Pointer2 is access all Integer;
   P1 : Pointer1;
   P2 : Pointer2;
   procedure Free is new Ada.Unchecked_Deallocation
      (Object => Integer, Name => Pointer2);
begin
   P1 := new Integer;
   P2 := Pointer2 (P1);
   Call_Something ( P2.all );
   ...
   Free (P2);
```

In the above, `P1.all` was allocated from `Pointer1'Storage_Pool`, but, via the type conversion, the code above is attempting to return it to `Pointer2'Storage_Pool`, which may be a different pool.

4.3.5 Compliant Code Example

```
   type Pointer1 is access all Integer;
   type Pointer2 is access all Integer;
   P1 : Pointer1;
   P2 : Pointer2;
   procedure Free is new Ada.Unchecked_Deallocation
      (Object => Integer, Name => Pointer1);
begin
   P1 := new Integer;
   P2 := Pointer2 (P1);
   Call_Something ( P2.all );
   ...
   Free (P1);
```

4.3.6 Notes

Enforcement of this rule can only be provided by manual code review, unless deallocation is forbidden via `No_Unchecked_Deallocation`.

CHAPTER

FIVE

CONCURRENCY (CON)

Goal

Maintainability
✓

Reliability
✓

Portability
✓

Performance
✓

Security

Description
Have a plan for managing the use of concurrency in high-integrity applications having real-time requirements.

Rules
CON01, CON02, CON03

The canonical approach to applications having multiple periodic and aperiodic activities is to map those activities onto independent tasks, i.e., threads of control. The advantages for the application are both a matter of software engineering and also ease of implementation. For example, when the different periods are not harmonics of one another, the fact that each task executes independently means that the differences are trivially represented. In contrast, such periods are not easily implemented in a cyclic scheduler, which, by definition, involves only one (implicit) thread of control with one frame rate.

High integrity applications are subject to a number of stringent analyses, including, for example, safety analyses and certification against rigorous industry standards. In addition, high integrity applications with real-time requirements must undergo timing analysis because they must be shown to meet deadlines prior to deployment — failure to meet hard deadlines is unacceptable in this domain.

These analyses are applied both to the application and to the implementation of the underlying run-time library. However, analysis of the complete set of general Ada tasking features is not tractable, neither technically nor in terms of cost. A subset of the language is required.

The Ravenscar profile [AdaRM2016] is a subset of the Ada concurrency facilities that supports determinism, schedulability analysis, constrained memory utilization, and certification to the highest integrity levels. Four distinct application domains are specifically intended:

- Hard real-time applications requiring predictability;
- Safety-critical systems requiring formal, stringent certification;
- High-integrity applications requiring formal static analysis and verification;

- Embedded applications requiring both a small memory footprint and low execution overhead.

Those tasking constructs that preclude analysis at the source level or analysis of the tasking portion of the underlying run-time library are disallowed.

The Ravenscar profile is necessarily strict in terms of what it removes so that it can support the stringent analyses, such as safety analysis, that go beyond the timing analysis required for real-time applications. In addition, the strict subset facilitates that timing analysis in the first place.

However, not all high-integrity applications are amenable to expression in the Ravenscar profile subset. The Jorvik profile [AdaRM2020] is an alternative subset of the Ada concurrency facilities. It is based directly on the Ravenscar profile but removes selected restrictions in order to increase expressive power, while retaining analyzability and performance. As a result, typical idioms for protected objects can be used, for example, and relative delays statements are allowed. Timing analysis is still possible but slightly more complicated, and the underlying run-time library is slightly larger and more complex.

When the most stringent analyses are required and the tightest timing is involved, use the Ravenscar profile. When a slight increase in complexity is tolerable, i.e., in those cases not undergoing all of these stringent analyses, consider using the Jorvik profile.

5.1 Use the Ravenscar Profile (CON01)

Level → Advisory

Category

Safety
✓

Cyber
✓

Goal

Maintainability
✓

Reliability
✓

Portability
✓

Performance
✓

Security

Remediation → High

Verification Method → GNATcheck rule: `uses_profile:ravenscar`

Mutually Exclusive → CON02

5.1.1 Reference

Ada Reference Manual: D.13 The Ravenscar and Jorvik Profiles[6]

5.1.2 Description

The following profile must be in effect:

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

The profile is equivalent to the following set of pragmas:

```
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);
```

5.1.3 Applicable Vulnerability within ISO TR 24772-2

- 6.59 Concurrency - Activation [GGA]
- 6.60 Concurrency - Directed termination [CGT]
- 6.61 Concurrent data access [CGX]
- 6.62 Concurrency - Premature termination [CGS]
- 6.63 Lock protocol errors [CGM]

[6] http://www.ada-auth.org/standards/12rm/html/RM-D-13.html

5.1.4 Noncompliant Code Example

Any code disallowed by the profile. Remediation is **high** because use of the facilities outside the subset can be difficult to retrofit into compliance.

```
task body Task_T is
begin
   loop
      -- Error: No_Relative_Delay
      delay 1.0;
      Put_Line ("Hello World");
   end loop;
end Task_T;
```

5.1.5 Compliant Code Example

```
task body Task_T is
   Period     : constant Time_Span := Milliseconds (10);
   Activation : Time               := Clock;
begin
   loop
      delay until Activation;
      Put_Line ("Hello World");
      Activation := Activation + Period;
   end loop;
end Task_T;
```

5.1.6 Notes

The Ada builder will detect violations if the programmer specifies this profile or corresponding pragmas. GNATcheck also can detect violations of profile restrictions.

5.2 Use the Jorvik Profile (CON02)

Level → Advisory

Category

Safety
✓

Cyber
✓

Goal

Maintainability
✓

Reliability
✓

Portability
✓

Performance
✓

Security

Remediation → High

Verification Method → GNATcheck rule: `uses_profile:jorvik`

Mutually Exclusive → CON01

5.2.1 Reference

Ada Reference Manual: D.13 The Ravenscar and Jorvik Profiles[7]

5.2.2 Description

The following profile must be in effect:

```
pragma Profile (Jorvik);
```

The profile is equivalent to the following set of pragmas:

```
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_Local_Protected_Objects,
           No_Local_Timing_Events,
           No_Protected_Type_Allocators,
           No_Requeue_Statements,
           No_Select_Statements,
           No_Specific_Termination_Handlers,
           No_Task_Allocators,
           No_Task_Hierarchy,
           No_Task_Termination,
           Pure_Barriers,
           Max_Task_Entries => 0,
           No_Dependence => Ada.Asynchronous_Task_Control,
           No_Dependence => Ada.Execution_Time.Group_Budgets,
           No_Dependence => Ada.Execution_Time.Timers,
           No_Dependence => Ada.Task_Attributes,
           No_Dependence => System.Multiprocessors.Dispatching_Domains);
```

The following restrictions are part of the Ravenscar profile but **not** part of the Jorvik profile.

```
No_Implicit_Heap_Allocations
No_Relative_Delay
Max_Entry_Queue_Length => 1
Max_Protected_Entries => 1
No_Dependence => Ada.Calendar
No_Dependence => Ada.Synchronous_Barriers
```

Jorvik also replaces restriction `Simple_Barriers` with `Pure_Barriers` (a weaker requirement than the restriction `Simple_Barriers`).

[7] http://www.ada-auth.org/standards/12rm/html/RM-D-13.html

5.2.3 Applicable Vulnerability within ISO TR 24772-2

- 6.59 Concurrency - Activation [GGA]
- 6.60 Concurrency - Directed termination [CGT]
- 6.61 Concurrent data access [CGX]
- 6.62 Concurrency - Premature termination [CGS]
- 6.63 Lock protocol errors [CGM]

5.2.4 Noncompliant Code Example

Any code disallowed by the profile. Remediation is **high** because use of the facilities outside the subset can be difficult to retrofit into compliance.

```
task body Task_T is
begin
   -- Error: Max_Task_Entries => 0
   accept Entry_Point do
      Put_Line ("Hello World");
   end Entry_Point;
   loop
      delay 1.0;
      Put_Line ("Ping");
   end loop;
end Task_T;
```

5.2.5 Compliant Code Example

```
task body Task_T is
begin
   delay 1.0;
   Put_Line ("Hello World");
   loop
      delay 1.0;
      Put_Line ("Ping");
   end loop;
end Task_T;
```

5.2.6 Notes

The Ada builder will detect violations. GNATcheck can also detect violations.

5.3 Avoid Shared Variables for Inter-task Communication (CON03)

Level → Advisory

Category

Safety
✓

Cyber
✓

Goal

Maintainability
✓

Reliability
✓

Portability
✓

Performance
✓

Security

Remediation → High

Verification Method → GNATcheck rule: `Volatile_Objects_Without_Address_Clauses`

5.3.1 Reference

Ada Reference Manual: D.13 The Ravenscar Profile[8]

5.3.2 Description

Although the Ravenscar and Jorvik profiles allow the use of shared variables for inter-task communication, such use is less robust and less reliable than encapsulating shared variables within protected objects.

5.3.3 Applicable Vulnerability within ISO TR 24772-2

- 6.56 Undefined behaviour [EWF]

5.3.4 Noncompliant Code Example

```
Global_Object : Integer
  with Volatile;
function Get return Integer is (Global_Object);
```

Note that variables marked as `Atomic` are also `Volatile`, per the Ada Reference Manual: C.6 (8/3) Shared Variable Control[9]

[8] http://www.ada-auth.org/standards/12rm/html/RM-D-13.html
[9] http://www.ada-auth.org/standards/12rm/html/RM-C-6.html

5.3.5 Compliant Code Example

When assigned to a memory address, a `Volatile` variable can be used to interact with a memory-mapped device, among other similar usages.

```
Global_Object : Integer
  with Volatile,
       Address => To_Address (16#1234_5678#);
function Get return Integer is (Global_Object);
```

5.3.6 Notes

In additon to GNATcheck, SPARK and CodePeer can also detect conflicting access to unprotected variables.

CHAPTER

SIX

ROBUST PROGRAMMING PRACTICE (RPP)

Goal

Maintainability
✓

Reliability
✓

Portability
✓

Performance
✓

Security
✓

Description
These rules promote the production of robust software.

Rules
RPP01, RPP02, RPP03, RPP04, RPP05, RPP06, RPP07, RPP07, RPP08, RPP09, RPP10, RPP11, RPP12, RPP13, RPP14

6.1 No Use of "others" in Case Constructs (RPP01)

Level → Required

Category

Safety
✓

Cyber
✓

Goal

Maintainability
✓

Reliability
✓

Portability
✓

Performance
Security

Remediation → Low

Verification Method → GNATcheck rule: `OTHERS_In_CASE_Statements`

6.1.1 Reference

[SEI-C] MSC01-C

6.1.2 Description

Case statement alternatives and case-expressions must not include use of the **others** discrete choice option. This rule prevents accidental coverage of a choice added after the initial case statement is written, when an explicit handler was intended for the addition.

Note that this is opposite to typical C guidelines such as [SEI-C] MSC01-C. The reason is that in C, the **default** alternative plays the role of defensive code to mitigate the switch statement's non-exhaustivity. In Ada, the **case** construct is exhaustive: the compiler statically verifies that for every possible value of the **case** expression there is a branch alternative, and there is also a dynamic check against invalid values which serves as implicit defensive code. As a result, Ada's **others** alternative doesn't play C's defensive code role and therefore a stronger guideline can be adopted.

6.1.3 Applicable Vulnerability within ISO TR 24772-2

- 6.27 Switch statements and static analysis [CLL]

6.1.4 Noncompliant Code Example

```
case Digit_T (C) is
   when '0' | '9' =>
      C := Character'succ (C);
   when others =>
      C := Character'pred (C);
end case;
```

6.1.5 Compliant Code Example

```
case Digit_T (C) is
   when '0' | '9' =>
      C := Character'succ (C);
   when '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' =>
      C := Character'pred (C);
end case;
```

6.1.6 Notes

N/A

6.2 No Enumeration Ranges in Case Constructs (RPP02)

Level → Required

Category

Safety
✓

Cyber
✓

Goal

Maintainability
✓

Reliability
✓

Portability
✓

Performance
Security

Remediation → Low

Verification Method → GNATcheck rule: `Enumeration_Ranges_In_CASE_Statements`

6.2.1 Reference

Similar to RPP01

6.2.2 Description

A range of enumeration literals must not be used as a choice in a **`case`** statement or a **`case`** expression. This includes explicit ranges (`A .. B`), subtypes, and the `'Range` attribute. Much like the use of **`others`** in **`case`** statement alternatives, the use of ranges makes it possible for a new enumeration value to be added but not handled with a specific alternative, when a specific alternative was intended.

6.2.3 Applicable Vulnerability within ISO TR 24772-2

- 6.5 Enumerator issues [CCB]

6.2.4 Noncompliant Code Example

```
case Digit_T (C) is
   when '0' | '9' =>
      C := Character'Succ (C);
   when '1' .. '8' =>
      C := Character'Pred (C);
end case;
```

6.2.5 Compliant Code Example

```
case Digit_T (C) is
   when '0' | '9' =>
      C := Character'Succ (C);
   when '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' =>
      C := Character'Pred (C);
end case;
```

6.2.6 Notes

N/A

6.3 Limited Use of "others" in Aggregates (RPP03)

Level → Advisory

Category

Safety
✓

Cyber
✓

Goal

Maintainability
✓

Reliability
✓

Portability
✓

Performance
Security

Remediation → Low

Verification Method → GNATcheck rule: `OTHERS_In_Aggregates`

6.3.1 Reference

Similar to RPP01

6.3.2 Description

Do not use an **others** choice in an extension aggregate. In **record** and **array** aggregates, do not use an **others** choice unless it is used either to refer to all components, or to all but one component.

This guideline prevents accidental provision of a general value for a **record** component or **array** component, when a specific value was intended. This possibility includes the case in which new components are added to an existing composite type.

6.3.3 Applicable Vulnerability within ISO TR 24772-2

- 6.5 Enumerator issues [CCB]
- 6.27 Switch statements and static analysis [CLL]

6.3.4 Noncompliant Code Example

```
type Record_T is record
   Field1 : Integer   := 1;
   Field2 : Boolean   := False;
   Field3 : Character := ' ';
end record;
type Array_T is array (Character) of Boolean;
Rec : Record_T := (Field1 => 1,
                   Field3 => '2',
                   others => <>);
Arr : Array_T := ('0' .. '9' => True,
                  others     => False);
```

6.3.5 Compliant Code Example

```
type Record_T is record
   Field1 : Integer   := 1;
   Field2 : Boolean   := False;
   Field3 : Character := ' ';
end record;
type Array_T is array (Character) of Boolean;
Rec : Record_T := (Field1 => 1,
                   others => <>);
Arr : Array_T := (others => False);
```

6.3.6 Notes

N/A

6.4 No Unassigned Mode-Out Procedure Parameters (RPP04)

Level → Required

Category

Safety
✓

Cyber
✓

Goal

Maintainability
✓

Reliability
✓

Portability
✓

Performance
Security

Remediation → High

Verification Method → GNATcheck rule: `Unassigned_OUT_Parameters`

6.4.1 Reference

MISRA C Rule 9.1: "The value of an object with automatic storage duration shall not be read before it has been set."

6.4.2 Description

For any procedure, all formal parameters of mode **`out`** must be assigned a value if the procedure exits normally. This rule ensures that, upon a normal return, the corresponding actual parameter has a defined value. Ensuring a defined value is especially important for scalar parameters because they are passed by value, such that some value is copied out to the actual. These undefined values can be especially difficult to locate because evaluation of the actual parameter's value might not occur immediately after the call returns.

6.4.3 Applicable Vulnerability within ISO TR 24772-2

- 6.32 Passing parameters and return values [CSJ]

6.4.4 Noncompliant Code Example

```
for Value_T use
  (Alpha   => 2#1#,
   Baker   => 2#10#,
   Charlie => 2#100#,
   Dog     => 2#1000#,
   Invalid => 2#1111#);

procedure Noncompliant (Register  :     Character;
                        Registera : out Value_T;
                        Registerb : out Value_T) is
begin
   if Register = 'A' then
      Registera := Alpha;
   end if;
end Noncompliant;
```

In the above example, some value is copied back for an output parameter as specified by `Register`. The other parameter is not assigned, and on return the value copied to the actual parameter may not be a valid representation for a value of the type. (We give the enumeration values a non-standard representation for the sake of illustration, i.e., to make it more likely that the undefined value is not valid.)

6.4.5 Compliant Code Example

```
procedure Compliant (Register  :     Character;
                     Registera : out Value_T;
                     Registerb : out Value_T) is
begin
   Registera := Invalid;
   Registerb := Invalid;
   if Register = 'A' then
      Registera := Alpha;
   end if;
end Compliant;
```

6.4.6 Notes

The GNATcheck rule specified above only detects a trivial case of an unassigned variable and doesn't provide a guarantee that there is no uninitialized access. It is not a replacement for a rigorous check for uninitialized access provided by advanced static analysis tools such as SPARK and CodePeer.

Note that the GNATcheck rule does not check function parameters (as of Ada 2012 functions can have **out** parameters). As a result, the better choice is either SPARK or CodePeer.

6.5 No Use of "others" in Exception Handlers (RPP05)

Level → Required

Category

Safety
✓

Cyber
✓

Goal

Maintainability
✓

Reliability
✓

Portability
✓

Performance
Security

Remediation → Low

Verification Method → GNATcheck rule: `OTHERS_In_Exception_Handlers`

6.5.1 Reference

N/A

6.5.2 Description

Much like the situation with **others** in **case** statements and **case** expressions, the use of **others** in exception handlers makes it possible to omit an intended specific handler for an exception, especially a new exception added to an existing set of handlers. As a result, a subprogram could return normally without having applied any recovery for the specific exception occurrence, which is likely a coding error.

6.5.3 Applicable Vulnerability within ISO TR 24772-2

N/A

6.5.4 Noncompliant Code Example

```
procedure Noncompliant (X : in out Integer) is
begin
   X := X * X;
exception
   when others =>
      X := -1;
end Noncompliant;
```

6.5.5 Compliant Code Example

```
procedure Compliant (X : in out Integer) is
begin
   X := X * X;
exception
   when Constraint_Error =>
      X := -1;
end Compliant;
```

6.5.6 Notes

ISO TR 24772-2: 6.50.2 slightly contradicts this when applying exception handlers around calls to library routines:

- Put appropriate exception handlers in all routines that call library routines, including the catch-all exception handler `when others =>`
- Put appropriate exception handlers in all routines that are called by library routines, including the catch-all exception handler `when others =>`

ISO TR 24772-2 also recommends "All tasks should contain an exception handler at the outer level to prevent silent termination due to unhandled exceptions."

6.6 Avoid Function Side-Effects (RPP06)

Level → Advisory

Category

Safety
✓

Cyber
✓

Goal

Maintainability
✓

Reliability
✓

Portability
✓

Performance
Security

Remediation → Medium

Verification Method → Code inspection

6.6.1 Reference

MISRA C Rule 13.2: "The value of an expression and its persistent side effects shall be the same under all permitted evaluation orders."

6.6.2 Description

Functions cannot update an actual parameter or global variable.

A side effect occurs when evaluation of an expression updates an object. This rule applies to function calls, a specific form of expression.

Side effects enable one form of parameter aliasing (see below) and evaluation order dependencies. In general they are a potential point of confusion because the reader expects only a computation of a value.

There are useful idioms based on functions with side effects. Indeed, a random number generator expressed as a function must use side effects to update the seed value. So-called "memo" functions are another example, in which the function tracks the number of times it is called. Therefore, exceptions to this rule are anticipated but should only be allowed on a per-instance basis after careful analysis.

6.6.3 Applicable Vulnerability within ISO TR 24772-2

- 6.24 Side-effects and order of evaluation [SAM]

6.6.4 Noncompliant Code Example

```
Call_Count : Integer := 0;
function F return Boolean is
   Result : Boolean;
begin
   ...
   Call_Count := Call_Count + 1;
   return Result;
end F;
```

6.6.5 Compliant Code Example

Remove the update to `Call_Count`, or change the function into a procedure with a parameter for `Call_Count`.

6.6.6 Notes

Violations are detected by SPARK as part of a rule disallowing side effects on expression evaluation.

6.7 Functions Only Have Mode "in" (RPP07)

Level → Required

Category

Safety
✓

Cyber
✓

Goal

Maintainability
✓

Reliability
✓

Portability
✓

Performance
Security

Remediation → Low

Verification Method → GNATcheck rule: `function_out_parameters`

6.7.1 Reference

N/A

6.7.2 Description

Functions must have only mode **in**.

As of Ada 2012, functions are allowed to have the same modes as procedures. However, this can lead to side effects and aliasing.

This rule disallows all modes except mode **in** for functions.

6.7.3 Applicable Vulnerability within ISO TR 24772-2

- 6.24 Side-effects and order of evaluation [SAM]

6.7.4 Noncompliant Code Example

```
function Noncompliant (Value : in out Integer) return Integer is
begin
   if Value < Integer'last then
      Value := Value + 1;
   end if;
   return Value;
end Noncompliant;
```

6.7.5 Compliant Code Example

```
function Compliant (Value : Integer) return Integer is
begin
   return Value + 1;
end Compliant;
```

OR

```
procedure Compliant (Value : in out Integer) is
begin
   if Value < Integer'last then
      Value := Value + 1;
   end if;
end Compliant;
```

6.7.6 Notes

Violations are detected by SPARK.

6.8 Limit Parameter Aliasing (RPP08)

Level → Required

Category

Safety
✓

Cyber
✓

Goal

Maintainability
✓

Reliability
✓

Portability
✓

Performance
Security

Remediation → High

Verification Method → Code inspection

6.8.1 Reference

Ada Reference Manual: 6.2 Formal Parameter Modes[10]

SPARK Reference Manual: Anti-Aliasing[11]

6.8.2 Description

In software, an alias is a name which refers to the same object as another name. In some cases, it is an error in Ada to reference an object through a name while updating it through another name in the same subprogram. Most of these cases cannot be detected by a compiler. Even when not an error, the presence of aliasing makes it more difficult to understand the code for both humans and analysis tools, and thus it may lead to errors being introduced during maintenance.

This rule is meant to detect problematic cases of aliasing that are introduced through the actual parameters and between actual parameters and global variables in a subprogram call. It is a simplified version of the SPARK rule for anti-aliasing defined in SPARK Reference Manual, section 6.4.2: Anti-Aliasing[12].

A formal parameter is said to be immutable when the subprogram cannot modify its value or modify the value of an object by dereferencing a part of the parameter of access type (at any depth in the case of SPARK). In Ada and SPARK, this corresponds to either an anonymous access-to-constant parameter or a parameter of mode `in` and not of an access type. Otherwise, the formal parameter is said to be mutable.

A procedure call shall not pass two actual parameters which potentially introduce aliasing via parameter passing unless either:

- both of the corresponding formal parameters are immutable; or
- at least one of the corresponding formal parameters is immutable and is of a by-copy type that is not an access type.

If an actual parameter in a procedure call and a global variable referenced by the called procedure potentially introduce aliasing via parameter passing, then:

- the corresponding formal parameter shall be immutable; and
- if the global variable is written in the subprogram, then the corresponding formal parameter shall be of a by-copy type that is not an access type.

Where one of the rules above prohibits the occurrence of an object or any of its subcomponents as an actual parameter, the following constructs are also prohibited in this context:

- A type conversion whose operand is a prohibited construct;
- A call to an instance of `Unchecked_Conversion` whose operand is a prohibited construct;
- A qualified expression whose operand is a prohibited construct;
- A prohibited construct enclosed in parentheses.

[10] http://www.ada-auth.org/standards/12rm/html/RM-6-2.html
[11] https://docs.adacore.com/spark2014-docs/html/lrm/subprograms.html#anti-aliasing
[12] https://docs.adacore.com/spark2014-docs/html/lrm/subprograms.html#anti-aliasing

6.8.3 Applicable Vulnerability within ISO TR 24772-2

- 6.32 Passing parameters and return values [CSJ]

6.8.4 Noncompliant Code Example

```
   type R is record
     Data : Integer := 0;
   end record;

   procedure Detect_Aliasing (Val_1 : in out R;
                              Val_2 : in R)
   is
   begin
      null;
   end Detect_Aliasing;

   Obj : R;

begin
   Detect_Aliasing (Obj, Obj);
```

6.8.5 Compliant Code Example

Do not pass `Obj` as the actual parameter to both formal parameters.

6.8.6 Notes

All violations are detected by SPARK. The GNAT compiler switch `-gnateA[1]` enables detection of some cases, but not all.

6.9 Use Precondition and Postcondition Contracts (RPP09)

Level → Advisory

Category

Safety
✓

Cyber
✓

Goal

Maintainability
✓

Reliability
✓

Portability
✓

	Performance	Security
		✓

Remediation → Low

Verification Method → Code inspection

6.9.1 Reference

Power of Ten rule 5: "The assertion density of the code should average to a minimum of two assertions per function."

6.9.2 Description

Subprograms should declare `Pre` and/or `Post` contracts. Developers should consider specifying the Global contract as well, when the default does not apply.

Subprogram contracts complete the Ada notion of a specification, enabling clients to know what the subprogram does without having to know how it is implemented.

Preconditions define those logical (Boolean) conditions required for the body to be able to provide the specified behavior. As such, they are obligations on the callers. These conditions are checked at run-time in Ada, prior to each call, and verified statically in SPARK.

Postconditions define those logical (Boolean) conditions that will hold after the call returns normally. As such, they express obligations on the implementer, i.e., the subprogram body. The implementation must be such that the postcondition holds, either at run-time for Ada, or statically in SPARK.

Not all subprograms will have both a precondition and a postcondition, some will have neither.

The `Global` contract specifies interactions with those objects not local to the corresponding subprogram body. As such, they help complete the specification because, otherwise, one would need to examine the body of the subprogram itself and all those it calls, directly or indirectly, to know whether any global objects were accessed.

6.9.3 Applicable Vulnerability within ISO TR 24772-2

- 6.42 Violations of the Liskov substitution principle or the contract model [BLP]

6.9.4 Noncompliant Code Example

```
type Stack is private;
procedure Push (This : in out Stack;  Item : Element);
```

6.9.5 Compliant Code Example

```
type Stack is private;
procedure Push (This : in out Stack;  Item : Element) with
   Pre  => not Full (This),
   Post => not Empty (This)
           and Top_Element (This) = Item
           and Extent (This) = Extent (This)'Old + 1
           and Unchanged (This'Old, Within => This),
   Global => null;
```

6.9.6 Notes

This rule must be enforced by manual inspection.

Moreover, the program must be compiled with enabled assertions (GNAT `-gnata` switch) to ensure that the contracts are executed, or a sound static analysis tool such as CodePeer or SPARK toolset should be used to prove that the contracts are always true.

6.10 Do Not Re-Verify Preconditions in Subprogram Bodies (RPP10)

Level → Advisory

Category

Safety
✓

Cyber
✓

Goal

Maintainability
✓

Reliability
✓

Portability
✓

Performance
Security

Remediation → Low

Verification Method → Static analysis tools

6.10.1 Reference

N/A

6.10.2 Description

Do not re-verify preconditions in the corresponding subprogram bodies. It is a waste of cycles and confuses the reader as well.

6.10.3 Applicable Vulnerability within ISO TR 24772-2

N/A

6.10.4 Noncompliant Code Example

```
type Stack is private;
procedure Push (This : in out Stack;  Item : Element) with
   Pre  => not Full (This),
   Post => ...
...
procedure Push (This : in out Stack;  Item : Element) is
begin
   if Full (This) then  -- redundant check
      raise Overflow;
   end if;
   This.Top := This.Top + 1;
   This.Values (This.Top) := Item;
end Push;
```

6.10.5 Compliant Code Example

```
type Stack is private;
procedure Push (This : in out Stack;  Item : Element) with
   Pre  => not Full (This),
   Post => ...
...
procedure Push (This : in out Stack;  Item : Element) is
begin
   This.Top := This.Top + 1;
   This.Values (This.Top) := Item;
end Push;
```

6.10.6 Notes

This rule can be enforced by CodePeer or SPARK, via detection of dead code.

6.11 Always Use the Result of Function Calls (RPP11)

Level → Advisory

Category

Safety
✓

Cyber
✓

Goal

Maintainability
✓

Reliability
✓

Portability
✓

Performance
Security

Remediation → Low

Verification Method → Compiler restrictions

6.11.1 Reference

MISRA C Rule 17.7: "The value returned by a function having non-void return type shall be used," and

Directive 4.7: "If a function returns error information, that error information shall be tested."

6.11.2 Description

In Ada and SPARK, it is not possible to ignore the object returned by a function call. The call must be treated as a value, otherwise the compiler will reject the call. For example, the value must be assigned to a variable, or passed as the actual parameter to a formal parameter of another call, and so on.

However, that does not mean that the value is actually used to compute some further results. Although almost certainly a programming error, one could call a function, assign the result to a variable (or constant), and then not use that variable further.

Note that functions will not have side-effects (due to RPP06) so it is only the returned value that is of interest here.

6.11.3 Applicable Vulnerability within ISO TR 24772-2

- 6.47 Inter-language calling [DJS]

6.11.4 Noncompliant Code Example

N/A

6.11.5 Compliant Code Example

N/A

6.11.6 Notes

The GNAT compiler warning switch `-gnatwu` (or the more general `-gnatwa` warnings switch) will cause the compiler to detect variables assigned but not read. CodePeer will detect these unused variables as well. SPARK goes further by checking that all computations contribute all the way to subprogram outputs.

6.12 No Recursion (RPP12)

Level → Advisory

Category

Safety
✓

Cyber
✓

Goal

Maintainability
✓

Reliability
✓

Portability
✓

Performance
Security

Remediation → Low

Verification Method → GNATcheck rule: `Recursive_Subprograms`

6.12.1 Reference

MISRA C Rule 17.2: "Functions shall not call themselves, either directly or indirectly."

6.12.2 Description

No subprogram shall be invoked, directly or indirectly, as part of its own execution.

In addition to making static analysis more complex, recursive calls make static stack usage analysis extremely difficult, requiring, for example, manual supply of call limits.

6.12.3 Applicable Vulnerability within ISO TR 24772-2

- 6.35 Recursion [GDL]

6.12.4 Noncompliant Code Example

```
function Noncompliant (N : Positive) return Positive is
begin
   if N = 1 then
      return 1;
   else
      return N * Noncompliant (N - 1);  -- could overflow
   end if;
end Noncompliant;
```

6.12.5 Compliant Code Example

```
function Compliant (N : Positive) return Positive is
   Result : Positive := 1;
begin
   for K in 2 .. N loop
      Result := Result * K;  -- could overflow
   end loop;
   return Result;
end Compliant;
```

6.12.6 Notes

The compiler will detect violations with the restriction `No_Recursion` in place. Note this is a dynamic check.

The GNATcheck rule specified above is a static check, subject to the limitations described in GNATcheck Reference Manual: Recursive Subprograms[13].

[13] https://docs.adacore.com/live/wave/lkql/html/gnatcheck_rm/gnatcheck_rm/predefined_rules.html#recursive-subprograms

6.13 No Reuse of Standard Typemarks (RPP13)

Level → Advisory

Category

Safety
✓

Cyber
✓

Goal

Maintainability
✓

Reliability
✓

Portability
✓

Performance
Security

Remediation → Low

Verification Method → GNATcheck rule: `overrides_standard_name`

6.13.1 Reference

N/A

6.13.2 Description

Do not reuse the names of standard Ada typemarks (e.g. `type Integer is range -1_000 .. 1_000;`)

When a developer uses an identifier that has the same name as a standard typemark, such as **`Integer`**, a subsequent maintainer might be unaware that this identifier does not actually refer to `Standard.`**`Integer`** and might unintentionally use the locally-scoped **`Integer`** rather than the original `Standard.`**`Integer`**. The locally-scoped **`Integer`** can have different attributes (and may not even be of the same base type).

6.13.3 Applicable Vulnerability within ISO TR 24772-2

N/A

6.13.4 Noncompliant Code Example

```
type Boolean is range 0 .. 1 with Size => 1;
type Character is ('A', 'E', 'I', 'O', 'U');
```

6.13.5 Compliant Code Example

```
type Boolean_T is range 0 .. 1 with Size => 1;
type Character_T is ('A', 'E', 'I', 'O', 'U');
```

6.13.6 Notes

N/A

6.14 Use Symbolic Constants for Literal Values (RPP14)

Level → Advisory

Category

Safety
✓

Cyber
✓

Goal

Maintainability
✓

Reliability
✓

Portability
✓

Performance
Security

Remediation → Low

Verification Method → GNATcheck rule: `Numeric_Literals`

6.14.1 Reference

N/A

6.14.2 Description

Extensive use of literals in a program can lead to two problems. First, the meaning of the literal is often obscured or unclear from the context. Second, changing a frequently used literal requires searching the entire program source for that literal and distinguishing the uses that must be modified from those that should remain unmodified.

Avoid these problems by declaring objects with meaningfully named constants, setting their values to the desired literals, and referencing the constants instead of the literals throughout the program. This approach clearly indicates the meaning or intended use of each literal. Furthermore, should the constant require modification, the change is limited to the declaration; searching the code is unnecessary.

Some literals can be replaced with attribute values. For example, when iterating over an array, it is better to use `Array_Object'First .. Array_Object'Last` than using `1 .. Array_Object'Length`.

6.14.3 Applicable Vulnerability within ISO TR 24772-2

N/A

6.14.4 Noncompliant Code Example

```
type Array_T is array (0 .. 31) of Boolean;
function Any_Set (X : Array_T) return Boolean is
   (for some Flag in 0 .. 31 => X (Flag));
```

6.14.5 Compliant Code Example

```
Number_Of_Bits : constant := 32;
type Array_T is array (0 .. Number_Of_Bits - 1) of Boolean;
function Any_Set (X : Array_T) return Boolean is
   (for some Flag in X'Range => X (Flag));
```

6.14.6 Notes

N/A

CHAPTER

SEVEN

EXCEPTION USAGE (EXU)

Goal

Maintainability
✓

Reliability
✓

Portability
✓

Performance
✓

Security
✓

Description
Have a plan for managing the use of Ada exceptions at the application level.

Rules
EXU01, EXU02, EXU03, EXU04

Exceptions in modern languages present the software architect with a dilemma. On one hand, exceptions can increase integrity by allowing components to signal specific errors in a manner that cannot be ignored, and, in general, allow residual errors to be caught. (Although there should be no unexpected errors in high integrity code, there may be some such errors due, for example, to unforeseeable events such as radiation-induced single-event upsets.) On the other hand, unmanaged use of exceptions increases verification expense and difficulty, especially flow analysis, perhaps to an untenable degree. In that case overall integrity is reduced or unwarranted.

In addition, programming languages may define some system-level errors in terms of language-defined exceptions. Such exceptions may be unavoidable, at least at the system level. For example, in Ada, stack overflow is signalled with the language-defined `Storage_Error` exception. Other system events, such as bus error, may also be mapped to language-defined or vendor-defined exceptions.

Complicating the issue further is the fact that, if exceptions are completely disallowed, there will be no exception handling code in the underlying run-time library. The effects are unpredictable if any exception actually does occur.

Therefore, for the application software the system software architect must decide whether to allow exceptions at all, and if they are to be used, decide the degree and manner of their usage. At the system level, the architect must identify the exceptions that are possible and how they will be addressed.

7.1 Do Not Raise Language-Defined Exceptions (EXU01)

Level → Required

Category

Safety
✓

Cyber
✓

Goal

Maintainability
✓

Reliability
✓

Portability
✓

Performance
Security

Remediation → Low

Verification Method → GNATcheck rule: `Raising_Predefined_Exceptions`

7.1.1 Reference

[SEI-Java] ERR07-J

7.1.2 Description

In no case should the application explicitly raise a language-defined exception.

The Ada language-defined exceptions are raised implicitly in specific circumstances defined by the language standard. Explicitly raising these exceptions would be confusing to application developers. The potential for confusion increases as the exception is propagated up the dynamic call chain, away from the point of the `raise` statement, because this increases the number of paths and thus corresponding language-defined checks that could have been the cause.

7.1.3 Applicable Vulnerability within ISO TR 24772-2

N/A

7.1.4 Noncompliant Code Example

```
procedure Noncompliant (X : in out Integer) is
begin
   if X < Integer'Last / 2
   then
      X := X * 2;
   else
      raise Constraint_Error;
   end if;
end Noncompliant;
```

7.1.5 Compliant Code Example

```
procedure Compliant (X : in out Integer) is
begin
   if X < Integer'Last / 2
   then
      X := X * 2;
   else
      raise Math_Overflow;
   end if;
end Compliant;
```

7.1.6 Notes

N/A

7.2 No Unhandled Application-Defined Exceptions (EXU02)

Level → Required

Category

Safety
✓

Cyber
✓

Goal

Maintainability
✓

Reliability
✓

Portability
✓

Performance
Security

Remediation → Low

Verification Method → GNATcheck rule: `Unhandled_Exceptions`

7.2.1 Reference

N/A

7.2.2 Description

All application-defined exceptions must have at least one corresponding handler that is applicable. Otherwise, if an exception is raised, undesirable behavior is possible. The term *applicable* means that there is no dynamic call chain that can reach the active exception which does not also include a handler that will be invoked for that exception, somewhere in that chain.

When an unhandled exception occurs in the sequence of statements of an application task and propagates to task's body, the task terminates abnormally. No *notification* of some sort is required or defined by the language, although some vendors' implementations may print out a log message or provide some other non-standard response. (Note that such a notification implies an external persistent environment, such as an operating system, that may not be present in all platforms.) The task failure does not affect any other tasks unless those other tasks attempt to communicate with it. In short, failure is silent.

Although the language-defined package `Ada.Task_Termination` can be used to provide a response using standard facilities, not all run-time libraries provide that package. For example, under the Ravenscar profile, application tasks are not intended to terminate, neither normally nor abnormally, and the language does not define what happens if they do. A run-time library for a memory-constrained target, especially a bare-metal target without an operating system, might not include any support for task termination when the tasking model is Ravenscar. The effects of task termination in that case are not defined by the language.

When an unhandled exception occurrence reaches the main subprogram and is not handled there, the exception occurrence is propagated to the environment task, which then completes abnormally. Even if the main subprogram does handle the exception, the environment task still completes (normally in that case).

When the environment task completes (normally or abnormally) it waits for the completion of dependent application tasks, if any. Those dependent tasks continue executing normally, i.e., they do not complete as a result of the environment task completion. Alternatively, however, instead of waiting for them, the implementation has permission to abort the dependent application tasks, per Ada Reference Manual: 10.2 (30) Program Execution[14] The resulting application-specific effect is undefined.

Finally, whether the environment task waited for the dependent tasks or aborted them, the semantics of further execution beyond that point are undefined. There is no concept of a calling environment beyond the environment task (Ada Reference Manual: 10.2 (30) Program Execution[15]). In some systems there is no calling environment, such as bare-metal platforms with only an Ada run-time library and no operating system.

[14] http://www.ada-auth.org/standards/12rm/html/RM-10-2.html
[15] http://www.ada-auth.org/standards/12rm/html/RM-10-2.html

7.2.3 Applicable Vulnerability within ISO TR 24772-2

- 6.36 Ignored error status and unhandled exceptions [OYB]

7.2.4 Noncompliant Code Example

```
procedure Main is
begin
   if Argument_Count = 0 then
      raise Cli_Exception;
   else
      begin
         Start_Application (Argument (1));
      exception
         when Application_Exception =>
            Put_Line ("Application failed");
      end;
   end if;
end Main;
```

7.2.5 Compliant Code Example

```
procedure Main is
begin
   if Argument_Count = 0 then
      raise Cli_Exception;
   else
      begin
         Start_Application (Argument (1));
      exception
         when Application_Exception =>
            Put_Line ("Application failed");
      end;
   end if;
exception
   when Cli_Exception =>
      Put_Line ("Failure");
end Main;
```

7.2.6 Notes

SPARK can prove that no exception will be raised (or fail to prove it and indicate the failure).

7.3 No Exception Propagation Beyond Name Visibility (EXU03)

Level → Required

Category

Safety

✓

Cyber
✓

Goal

Maintainability
✓

Reliability
✓

Portability
✓

Performance
Security

Remediation → Low

Verification Method → GNATcheck rule: `Non_Visible_Exceptions`

7.3.1 Reference

RPP05

7.3.2 Description

An active exception can be propagated dynamically past the point where the name of the exception is visible (the scope of the declaration). The exception can only be handled via **others** past that point. That situation prevents handling the exception specifically, and violates RPP05.

7.3.3 Applicable Vulnerability within ISO TR 24772-2

N/A

7.3.4 Noncompliant Code Example

```
procedure Noncompliant (Param : in out Integer) is
   Noncompliant_Exception : exception;
begin
   Param := Param * Param;
exception
   when others =>
      raise Noncompliant_Exception;
end Noncompliant;
```

As a result the exception name cannot be referenced outside the body:

```
procedure Bad_Call (Param : in out Integer) is
begin
   Noncompliant (Param);
exception
   when Noncompliant_Exception =>  -- compile error
      null;
end Bad_Call;
```

7.3.5 Compliant Code Example

```
Compliant_Exception : exception;
procedure Compliant (Param : in out Integer) is
begin
   Param := Param * Param;
exception
   when others =>
      raise Compliant_Exception;
end Compliant;

procedure Good_Call (Param : in out Integer) is
begin
   Compliant (Param);
exception
   when Compliant_Exception =>
      null;
end Good_Call;
```

7.3.6 Notes

N/A

7.4 Prove Absence of Run-time Exceptions (EXU04)

Level → Required

Category

Safety
✓

Cyber
✓

Goal

Maintainability
✓

Reliability
✓

Portability
✓

Performance
Security

Remediation → Low

Verification Method → Compiler restrictions

7.4.1 Reference

MISRA C Rule 1.3: "There shall be no occurrence of undefined or critical unspecified behaviour."

7.4.2 Description

In many high-integrity systems the possible responses to an exception are limited or nonexistent. In these cases the only approach is to prove exceptions cannot occur in the first place. Additionally, the cost of proving exceptions cannot happen may be less than the cost of analyzing code in which they are allowed to be raised.

The restriction `No_Exceptions` can be used with **`pragma`** *`Restrictions`* to enforce this approach. Specifically, the restriction ensures that **`raise`** statements and exception handlers do not appear in the source code and that language-defined checks are not emitted by the compiler. However, a run-time check performed automatically by the hardware is permitted because it typically cannot be prevented. An example of such a check would be traps on invalid addresses. If a hardware check fails, or if an omitted language-defined check would have failed, execution is unpredictable. As a result, enforcement with the restriction is not ideal. However, proof of the absence of run-time errors is possible using the SPARK subset of Ada.

7.4.3 Applicable Vulnerability within ISO TR 24772-2

N/A

7.4.4 Noncompliant Code Example

N/A

7.4.5 Compliant Code Example

N/A

7.4.6 Notes

This restriction is detected by SPARK, in which any statements explicitly raising an exception must be proven unreachable (or proof fails and the failure is indicated), and any possibility of run-time exception should be proved not to happen.

CHAPTER

EIGHT

OBJECT-ORIENTED PROGRAMMING (OOP)

Goal

Maintainability
✓

Reliability
✓

Portability
Performance
Security
✓

Description
Have a plan for selecting the OOP facilities of the language to use.

Rules
OOP01, OOP02, OOP03, OOP04, OOP05, OOP06, OOP07

There are many issues to consider when planning the use of Object Oriented features in a high-integrity application. Choices should be made based on the desired expressive power of the OO features and the required level of certification or safety case.

For example, the use of inheritance can provide abstraction and separation of concerns. However, the extensive use of inheritance, particularly in deep hierarchies, can lead to fragile code bases.

Similarly, when new types of entities are added, dynamic dispatching provides separation of the code that must change from the code that manipulates those types and need not be changed to handle new types. However, analysis of dynamic dispatching must consider every candidate object type and analyze the associated subprogram for appropriate behavior.

Therefore, the system architect has available a range of possibilities for the use of OOP constructs, with tool enforcement available for the selections. Note that full use of OOP, including dynamic dispatching, may not be unreasonable.

The following rules assume use of tagged types, a requirement for full OOP in Ada.

8.1 No Class-wide Constructs Policy (OOP01)

Level → Advisory

Category

Safety
✓

Cyber
✓

Goal

Maintainability
✓

Reliability
✓

Portability
Performance
Security
✓

Remediation → N/A

Verification Method → Compiler restrictions

Mutually Exclusive → OOP02

8.1.1 Reference

N/A

8.1.2 Description

In this approach, tagged types are allowed and type extension (inheritance) is allowed, but there are no class-wide constructs.

This restriction ensures there are no class-wide objects or formal parameters, nor access types designating class-wide types.

In this approach there are no possible dynamic dispatching calls because such calls can only occur when a class-wide value is passed as the parameter to a primitive operation of a tagged type.

8.1.3 Applicable Vulnerability within ISO TR 24772-2

- 6.43 Redispatching [PPH]

8.1.4 Noncompliant Code Example

```
X : Object'Class := Some_Object;
```

8.1.5 Compliant Code Example

```
X : Object := Some_Object;
```

8.1.6 Notes

The compiler will detect violations with the standard Ada restriction `No_Dispatch` applied.

8.2 Static Dispatching Only Policy (OOP02)

Level → Advisory

Category

Safety
✓

Cyber
✓

Goal

Maintainability
✓

Reliability
✓

Portability
Performance
Security
✓

Remediation → N/A

Verification Method → Compiler restrictions

Mutually Exclusive → OOP01

8.2.1 Reference

N/A

8.2.2 Description

In this approach, class-wide constructs are allowed, as well as tagged types and type extension (inheritance), but dynamic dispatching remains disallowed (i.e., as in OOP01).

This rule ensures there are no class-wide values passed as the parameter to a primitive operation of a tagged type, hence there are no dynamically dispatched calls.

Note that this rule should not be applied without due consideration.

8.2.3 Applicable Vulnerability within ISO TR 24772-2

- 6.43 Redispatching [PPH]

8.2.4 Noncompliant Code Example

```
Some_Primitive (Object'Class (X));
```

8.2.5 Compliant Code Example

```
Some_Primitive (X);
```

8.2.6 Notes

N/A

8.3 Limit Inheritance Hierarchy Depth (OOP03)

Level → Advisory

Category

Safety
✓

Cyber
✓

Goal

Maintainability
✓

Reliability
✓

Portability
Performance
Security
✓

Remediation → High

Verification Method → GNATcheck rule: `Deep_Inheritance_Hierarchies:2`

8.3.1 Reference

[AdaOOP2016] section 5.1

8.3.2 Description

A class inheritance hierarchy consists of a set of types related by inheritance. Each class, other than the root class, is a subclass of other classes, and each, except for "leaf" nodes, is a base class for those that are derived from it.

Improperly designed inheritance hierarchies complicate system maintenance and increase the effort in safety certification, in any programming language.

A common characteristic of problematic hierarchies is "excessive" depth, in which a given class is a subclass of many other classes. Depth can be a problem because a change to a class likely requires inspection, modification, recompilation, and retesting/reverification of all classes below it in the hierarchy. The extent of that effect increases as we approach the root class. This rippling effect is known as the *fragile base class* problem. Clearly, the greater the depth the more subclasses there are to be potentially affected. In addition, note that a change to one class may cause a cascade of other secondary changes to subclasses, so the effect is often not limited to a single change made to all the subclasses in question.

Deep inheritance hierarchies also contribute to complexity, rather than lessening it, by requiring the reader to understand multiple superclasses in order to understand the behavior of a given subclass.

8.3.3 Applicable Vulnerability within ISO TR 24772-2

- 6.41 Inheritance [RIP]

8.3.4 Noncompliant Code Example

The threshold for "too deep" is inexact, but beyond around 4 or 5 levels the complexity accelerates rapidly.

```
type Shape_T is tagged private;
procedure Set_Name (Shape : Shape_T; Name  : String);
function Get_Name (Shape : Shape_T) return String;

type Quadrilateral_T is new Shape_T with private;
type Trapezoid_T is new Quadrilateral_T with private;
type Parallelogram_T is new Trapezoid_T with private;
type Rectangle_T is new Parallelogram_T with private;
type Square_T is new Rectangle_T with private;
```

8.3.5 Compliant Code Example

```
type Shape_T is tagged private;
procedure Set_Name (Shape : Shape_T; Name  : String);
function Get_Name (Shape : Shape_T) return String;

type Quadrilateral_T is new Shape_T with private;
type Rectangle_T is new Quadrilateral_T with private;
type Square_T is new Rectangle_T with private;
```

8.3.6 Notes

N/A

8.4 Limit Statically-Dispatched Calls to Primitive Operations (OOP04)

Level → Advisory

Category

Safety
✓

Cyber
✓

Goal

Maintainability
✓

Reliability
✓

Portability
✓

Performance
Security

Remediation → Medium (easy fix, but a difficult to detect bug)

Verification Method → GNATcheck rule: `Direct_Calls_To_Primitives`

8.4.1 Reference

N/A

8.4.2 Description

This rule applies only to tagged types, when visibly tagged at the point of a call from one primitive to another of that same type.

By default, subprogram calls are statically dispatched. Dynamic dispatching only occurs when a class-wide value is passed to a primitive operation of a specific type. Forcing an otherwise optional dynamic dispatching call in this case is known as *redispatching.*

When one primitive operation of a given tagged type invokes another distinct primitive operation of that same type, use redispatching so that an overriding version of that other primitive will be invoked if it exists. Otherwise an existing overridden version would not be invoked, which is very likely an error.

This rule does not apply to the common case in which an overriding of a primitive operation calls the "parent" type's version of the overridden operation. Such calls occur in the overridden body when the new version is not replacing, but rather, is augmenting the parent type's version. In this case the new version must do whatever the parent version did, and can then add functionality specific to the new type.

By default, this rule applies to another common case in which static calls from one primitive operation to another make sense. Specifically, *constructors* are often implemented in Ada as functions that create a new value of the tagged type. As constructors, these functions are type-specific. They must call the primitive operations of the type being created, not operations that may be overridden for some type later derived from it. (Note that there is a GNATcheck rule parameter to not flag this case.)

Typically constructor functions only have the tagged type as the result type, not as the type for formal parameters, if any, because actual parameters of the tagged type would themselves likely require construction. This specific usage is the case ignored by the GNATcheck rule parameter.

Note that constructors implemented as procedures also call primitive operations of the specific type, for the same reasons as constructor functions. This usage is allowed by this rule and does not require the GNATcheck parameter. (The difference between function and procedure constructors is that these procedures will have a formal parameter of the tagged type, of mode **out**.)

8.4.3 Applicable Vulnerability within ISO TR 24772-2

- 6.42 Violations of the Liskov substitution principle of the contract model [BLP]
- 6.43 Redispatching [PPH]
- 6.44 Polymorphic variables [BKK]

8.4.4 Noncompliant Code Example

Class constructs

```
package Root is
   type Root_T is tagged null record;
   procedure Noncompliant (X : in out Root_T) is null;
   procedure Compliant (X : in out Root_T) is null;
   procedure Other_Prim (X : in out Root_T) is null;
end Root;

package Child is
   use Root;
```

(continues on next page)

(continued from previous page)

```
   type Child_T is new Root_T with null record;
   procedure Noncompliant (X : in out Child_T);
   procedure Compliant (X : in out Child_T);
   procedure Other_Prim (X : in out Child_T);
end Child;

procedure Not_A_Primitive (X : in out Child.Child_T) is null;
```

Noncompliant Code

```
procedure Noncompliant (X : in out Child_T) is
begin
   Other_Prim (Root_T (X));
   Other_Prim (X);
end Noncompliant;
```

8.4.5 Compliant Code Example

```
procedure Compliant (X : in out Child_T) is
begin
   Compliant (Root_T (X)); -- constructor style is OK
   Not_A_Primitive (X);
end Compliant;
```

8.4.6 Notes

N/A

8.5 Use Explicit Overriding Annotations (OOP05)

Level → Required

Category

Safety
✓

Cyber
✓

Goal

Maintainability
✓

Reliability
✓

Portability
✓

Performance
Security

Remediation → Low

Verification Method → GNATcheck rule: `Style_Checks:0`

8.5.1 Reference

[AdaOOP2016] section 4.3

8.5.2 Description

The declaration of a primitive operation that overrides an inherited operation must include an explicit **overriding** annotation.

The semantics of inheritance in mainstream object-oriented languages may result in two kinds of programming errors: 1) intending, but failing, to override an inherited subprogram, and 2) intending not to override an inherited subprogram, but doing so anyway. Because an overridden subprogram may perform subclass-specific safety or security checks, the invocation of the parent subprogram on a subclass instance can introduce a vulnerability.

The first issue (intending but failing to override) typically occurs when the subprogram name is misspelled. In this case a new or overloaded subprogram is actually declared.

The second issue (unintended overriding) can arise when a new subprogram is added to a parent type in an existing inheritance hierarchy. The new subprogram happens to cause one or more inherited subprograms below it to override the new superclass version. This mistake typically happens during program maintenance.

In Ada, much like other modern languages, one can annotate a subprogram declaration (and body) with an indication that the subprogram is an overriding of an inherited version. This is done with the **overriding** reserved word preceding the subprogram specification.

Similarly, in Ada one can explicitly indicate that a subprogram is not an overriding. To do so, the programmer includes the reserved words **not overriding** immediately prior to the subprogram specification.

Of course, incorrect marking errors are flagged by the compiler. If a subprogram is explicitly marked as overriding but is not actually overriding, the compiler will reject the code. Likewise, if a primitive subprogram is explicitly marked as not overriding, but actually is overriding, the compiler will reject the code.

However, most subprograms are not overriding so it would be a heavy burden on the programmer to make them explicitly indicate that fact. That's not to mention the relatively heavy syntax required.

In addition, both annotations are optional for the sake of compatibility with prior versions of the language. Therefore, a subprogram without either annotation might or might not be overriding. A legal program could contain some explicitly annotated subprograms and some that are not annotated at all. But because the compiler will reject explicit annotations that are incorrect, all we require is that one of the two cases be explicitly indicated for all such subprograms. Any unannotated subprograms not flagged as errors are then necessarily not that case, they must be the other one.

Since overriding is less common and involves slightly less syntax to annotate, the guideline requires explicit annotations only for overriding subprograms. It follows that any subprograms not flagged as errors by the compiler are not overriding, so they need not be marked explicitly as such.

This guideline is implemented by compiler switches, or alternatively, by a GNATcheck rule (specified below the table). With this guideline applied and enforced, the two inheritance errors described in the introduction cannot happen.

Note that the compiler switches will also require the explicit overriding indicator when overriding a language-defined operator. The switches also apply to inherited primitive subprograms for non-tagged types.

8.5.3 Applicable Vulnerability within ISO TR 24772-2

- 6.34 Subprogram signature mismatch [OTR]
- 6.41 Inheritance [RIP]

8.5.4 Noncompliant Code Example

```
type Root_T is tagged null record;
procedure Primitive (X : in out Root_T) is null;
```

```
type Noncompliant_Child_T is new Root_T with null record;
procedure Primitive (X : in out Noncompliant_Child_T) is null;
```

8.5.5 Compliant Code Example

```
type Compliant_Child_T is new Root_T with null record;
overriding procedure Primitive (X : in out Compliant_Child_T) is null;
```

8.5.6 Notes

This rule requires the GNAT compiler switches `-gnaty0` and `-gnatwe` in order for the compiler to flag missing overriding annotations as errors. The first causes the compiler to generate the warnings, and the second causes those warnings to be treated as errors.

8.6 Use Class-wide Pre/Post Contracts (OOP06)

Level → Required

Category

Safety
✓

Cyber
✓

Goal

Maintainability
✓

Reliability
✓

Portability
Performance
Security
✓

Remediation → Low

Verification Method → GNATcheck rule: `Specific_Pre_Post`

8.6.1 Reference

[AdaOOP2016] section 6.1.4

SPARK User's Guide, section 7.5.2[16]

8.6.2 Description

For primitive operations of tagged types, use only class-wide pre/post contracts, if any.

The class-wide form of precondition and postcondition expresses conditions that are intended to apply to any version of the subprogram. Therefore, when a subprogram is derived as part of inheritance, only the class-wide form of those contracts is inherited from the parent subprogram, if any are defined. As a result, it only makes sense to use the class-wide form in this situation.

(The same semantics and recommendation applies to type invariants.)

Note: this approach will be required for OOP07 (Ensure Local Type Consistency).

8.6.3 Applicable Vulnerability within ISO TR 24772-2

- 6.42 Violations of the Liskov substitution principle or the contract model [BLP]

8.6.4 Noncompliant Code Example

```
type Root_T is tagged null record;
procedure Set_Name (X    : Root_T;
                    Name : String)
  with Pre => Name'length > 0;
function Get_Name (X : Root_T) return String
  with Post => Get_Name'result'length > 0;
```

8.6.5 Compliant Code Example

```
type Root_T is tagged null record;
procedure Set_Name (X    : Root_T;
                    Name : String)
  with Pre'class => Name'length > 0;
function Get_Name (X : Root_T) return String
  with Post'class => Get_Name'result'length > 0;
```

[16] https://docs.adacore.com/live/wave/spark2014/html/spark2014_ug/en/source/how_to_write_object_oriented_contracts.html#writing-contracts-on-dispatching-subprograms

8.6.6 Notes

N/A

8.7 Ensure Local Type Consistency (OOP07)

Level → Required

Category

Safety
✓

Cyber
✓

Goal

Maintainability
✓

Reliability
✓

Portability
Performance
Security
✓

Remediation → N/A

Verification Method → Software test

8.7.1 Reference

[AdaOOP2016] See section 4.2.

GNAT User's Guide, section 5.10.11[17]

8.7.2 Description

Either:

- Formally verify local type consistency, or
- Ensure that each tagged type passes all the tests of all the parent types which it can replace.

Rationale:

One of the fundamental benefits of OOP is the ability to manipulate objects in a class inheritance hierarchy without "knowing" at compile-time the specific classes of the objects being manipulated. By *manipulate* we mean invoking the primitive operations, the *methods* defined by the classes.

We will use the words *class* and *type* interchangeably, because classes are composed in Ada and SPARK using a combination of building blocks, especially type declarations. However, we will use the term *subclass* rather than *subtype* because the latter has a specific meaning in Ada and SPARK that is unrelated to OOP.

[17] https://docs.adacore.com/gnat_ugn-docs/html/gnat_ugn/gnat_ugn/gnat_utility_programs.html

The objects whose operations are being invoked can be of types anywhere in the inheritance tree, from the root down to the bottom. The location, i.e., the specific type, is transparent to the manipulating code. This type transparency is possible because the invoked operations are dynamically dispatched at run-time, rather than statically dispatched at compile-time.

Typically, the code manipulating the objects does so in terms of superclasses closer to the root of the inheritance tree. Doing so increases generality because it increases the number of potential subclasses that can be manipulated. The actual superclass chosen will depend on the operations required by the manipulation. In Ada and SPARK, subclasses can add operations but can never remove them, so more operations are found as we move down from the root. That is the nature of specialization. Note that the guarantee of an invoked operations' existence is essential for languages used in this domain.

However, for this transparent manipulation to be functionally correct — to accomplish what the caller intends — the primitive operations of subclasses must be functionally indistinguishable from those of the superclasses. That doesn't mean the subclasses cannot make changes. Indeed, the entire point of subclasses is to make changes. In particular, functional changes can be either introduction of new operations, or overridings of inherited operations. It is these overridings that must be functionally transparent to the manipulating code. (Of course, for an inherited operation that is not overridden, the functionality is inherited as-is, and is thus transparent trivially.)

The concept of functional transparency was introduced, albeit with different terminology, by Liskov and Wing in 1994 [LiskovWing1994] and is, therefore, known as the Liskov Substitution Principle, or LSP. The *substitution* in LSP means that a subclass must be substitutable for its superclass, i.e., a subclass instance should be usable whenever a superclass instance is required.

Unfortunately, requirements-based testing will not detect violations of LSP because unit-level requirements do not concern themselves with superclass substitutability.

However, the OO supplement of DO-178C [DO178C] offers solutions, two of which are in fact the options recommended by this guideline.

Specifically, the supplement suggests adding a specific verification activity it defines as Local Type Consistency Verification. This activity ensures LSP is respected and, in so doing, addresses the vulnerability.

Verification can be accomplished statically with formal methods in SPARK, or dynamically via a modified form of testing.

For the static approach, type consistency is verified by examining the properties of the overriding operation's preconditions and postconditions. These are the properties required by Design by Contract, as codified by Bertrand Meyer [Meyer1997]. Specifically, an overridden primitive may only replace the precondition with one weaker than that of the parent version, and may only replace the postcondition with one stronger. In essence, relative to the parent version, an overridden operation can only require the same or less, and provide the same or more. Satisfying that requirement is sufficient to ensure functional transparency because the manipulating code only "knows" the contracts of the class it manipulates, i.e., the view presented by the type, which may very well be a superclass of the one actually invoked.

In Ada and SPARK, the class-wide form of preconditions and postconditions are inherited by overridden primitive operations of tagged types. The inherited precondition and that of the overriding (if any) are combined into a conjunction. All must hold, otherwise the precondition fails. Likewise, the inherited postcondition is combined with the overriding postcondition into a disjunction. At least one of them must hold. In Ada these are tested at run-time. In SPARK, they are verified statically (or not, in which case proof fails and an error is indicated).

To verify substitutability via testing, all the tests for all superclass types are applied to objects of the given subclass type. If all the parent tests pass, this provides a high degree

of confidence that objects of the new tagged type can properly substitute for parent type objects. Note that static proof of consistency provides an even higher degree of confidence.

8.7.3 Applicable Vulnerability within ISO TR 24772-2

- 6.42 Violations of the Liskov substitution principle of the contract model [BLP]
- 6.43 Redispatching [PPH]
- 6.44 Polymorphic variables [BKK]

8.7.4 Noncompliant Code Example

```
package P is
   pragma Elaborate_Body;
   type Rectangle is tagged private;
   procedure Set_Width (This  : in out Rectangle;
                        Value : Positive)
   with
      Post => Width (This) = Value and
              Height (This) = Height (This'Old);

   function Width (This : Rectangle) return Positive;

   procedure Set_Height (This  : in out Rectangle;
                         Value : Positive)
   with
      Post => Height (This) = Value and
              Width (This) = Width (This'Old);

   function Height (This : Rectangle) return Positive;

private
   ...
end P;
```

The postcondition for `Set_Width` states that the `Height` is not changed. Likewise, for `Set_Height`, the postcondition asserts that the `Width` is not changed. However, these postconditions are not class-wide so they are not inherited by subclasses.

Now, in a subclass `Square`, the operations are overridden so that setting the width also sets the height to the same value, and vice versa. Thus the overridden operations do not maintain type consistency, but this fact is neither detected at run-time, nor could SPARK verify it statically (and SPARK is not used at all in these versions of the packages).

```
with P; use P;
package Q is
   pragma Elaborate_Body;
   type Square is new Rectangle with private;

   overriding
   procedure Set_Width (This  : in out Square;
                        Value : Positive)
   with
  Post => Width (This) = Height (This);

   overriding
   procedure Set_Height (This  : in out Square;
                         Value : Positive)
```

(continues on next page)

(continued from previous page)

```
   with
  Post  => Width (This) = Height (This);

private
   ...
end Q;
```

8.7.5 Compliant Code Example

```
package P with SPARK_Mode is
   pragma Elaborate_Body;
   type Rectangle is tagged private;

   procedure Set_Width (This  : in out Rectangle;
                        Value : Positive)
   with
      Post'Class => Width (This) = Value and
                    Height (This) = Height (This'Old);

   function Width (This : Rectangle) return Positive;

   procedure Set_Height (This  : in out Rectangle;
                         Value : Positive)
   with
      Post'Class => Height (This) = Value and
                    Width (This) = Width (This'Old);

   function Height (This : Rectangle) return Positive;

private
   ...
end P;
```

Now the postconditions are class-wide so they are inherited by subclasses. In the subclass Square, the postconditions will not hold at run-time. Likewise, SPARK can now prove that type consistency is not verified because the postconditions are weaker than those inherited:

```
with P; use P;
package Q with SPARK_Mode is
   pragma Elaborate_Body;
   type Square is new Rectangle with private;

   overriding
   procedure Set_Width (This  : in out Square;
                        Value : Positive)
   with
  Post'Class => Width (This) = Height (This);

   overriding
   procedure Set_Height (This  : in out Square;
                         Value : Positive)
   with
  Post'Class => Width (This) = Height (This);

private
   type Square is new Rectangle with null record;
end Q;
```

8.7.6 Notes

Verification can be achieved dynamically with the GNATtest tool, using the `--validate-type-extensions` switch. SPARK enforces this rule.

CHAPTER

NINE

SOFTWARE ENGINEERING (SWE)

Goal

Maintainability
✓

Reliability
✓

Portability
✓

Performance
Security
✓

Description
These rules promote "best practices" for software development.

Rules
SWE01, SWE02, SWE03, SWE04

9.1 Use SPARK Extensively (SWE01)

Level → Advisory

Category

Safety
✓

Cyber
✓

Goal

Maintainability
✓

Reliability
✓

Portability
✓

Performance
✓

Security
✓

Remediation → High, as retrofit can be extensive

Verification Method → Compiler restrictions

9.1.1 Reference

SPARK User's Guide, section 8: "Applying SPARK in Practice"[18]

9.1.2 Description

SPARK has proven itself highly effective, both in terms of low defects, low development costs, and high productivity. The guideline advises extensive use of SPARK, especially for the sake of formally proving the most critical parts of the source code. The rest of the code can be in SPARK as well, even if formal proof is not intended, with some parts in Ada when features outside the SPARK subset are essential.

9.1.3 Applicable Vulnerability within ISO TR 24772-2

N/A

9.1.4 Noncompliant Code Example

Any code outside the (very large) SPARK subset is flagged by the compiler.

9.1.5 Compliant Code Example

N/A

9.1.6 Notes

Violations are detected by the SPARK toolset.

9.2 Enable Optional Warnings and Treat As Errors (SWE02)

Level → Required

Category

Safety
✓

Cyber
✓

Goal

Maintainability
✓

[18] https://docs.adacore.com/live/wave/spark2014/html/spark2014_ug/en/usage_scenarios.html

Reliability
✓

Portability
Performance
Security
✓

Remediation → Low

Verification Method → Compiler restrictions

9.2.1 Reference

Power of 10 rule #10: "All code must be compiled, from the first day of development, with all compiler warnings enabled at the most pedantic setting available. All code must compile without warnings."

9.2.2 Description

The Ada compiler does a degree of static analysis itself, and generates many warnings when they are enabled. These warnings likely indicate very real problems so they should be examined and addressed, either by changing the code or disabling the warning for the specific occurrence flagged in the source code.

To ensure that warnings are examined and addressed one way or the other, the compiler must be configured to treat warnings as errors, i.e., preventing object code generation.

Note that warnings will occasionally be given for code usage that is intentional. In those cases the warnings should be disabled by using `pragma Warnings` with the parameter `Off`, and a string indicating the error message to be disabled. In other cases, a different mechanism might be appropriate, such as aspect (or pragma) `Unreferenced`.

9.2.3 Applicable Vulnerability within ISO TR 24772-2

- 6.18 Dead Store [WXQ]
- 6.19 Unused variable [YZS]
- 6.20 Identifier name reuse [YOW]
- 6.22 Initialization of variables [LAV]

9.2.4 Noncompliant Code Example

```
procedure P (This : Obj) is
begin
   ... code not referencing This
end P;
```

The formal parameter controls dispatching for the sake of selecting the subprogram to be called but does not participate in the implementation of the body.

9.2.5 Compliant Code Example

```
procedure P (This : Obj) is
   pragma Unreferenced (This);
begin
   ... code not referencing This
end P;
```

The compiler will no longer issue a warning that the formal parameter `This` is not referenced. Of course, if that changes and `This` becomes referenced, the compiler will flag the **`pragma`**.

9.2.6 Notes

This rule can be applied via the GNAT `-gnatwae` compiler switch, which both enables warnings and treats them as errors. Note that the switch enables almost all optional warnings, but not all. Some optional warnings correspond to very specific circumstances, and would otherwise generate too much noise for their value.

9.3 Use a Static Analysis Tool Extensively (SWE03)

Level → Mandatory

Category

Safety
✓

Cyber
✓

Goal

Maintainability
✓

Reliability
✓

Portability
✓

Performance
✓

Security
✓

Remediation → High

Verification Method → Static analysis tools

9.3.1 Reference

Power of 10 rule #10: "All code must also be checked daily with at least one, but preferably more than one, strong static source code analyzer and should pass all analyses with zero warnings."

9.3.2 Description

If not using SPARK for regular development, use a static analyzer, such as CodePeer, extensively. No warnings or errors should remain unresolved at the given level adopted for analysis (which can be selected to adjust the false positive ratio).

Specifically, any code checked into the configuration management system must be checked by the analyzer and be error-free prior to check-in. Similarly, each nightly build should produce a CodePeer baseline for the project.

9.3.3 Applicable Vulnerability within ISO TR 24772-2

- 6.6 Conversion errors [FLC]
- 6.18 Dead store [WXQ]
- 6.19 Unused variable [YZS]
- 6.20 Identifier name reuse [YOW]
- 6.24 Side-effects and order of evaluation [SAM]
- 6.25 Likely incorrect expression [KOA]

9.3.4 Noncompliant Code Example

N/A

9.3.5 Compliant Code Example

N/A

9.3.6 Notes

CodePeer is the recommended static analyzer. Note that CodePeer can detect GNATcheck rule violations (via the `--gnatcheck` CodePeer switch and a rules file).

9.4 Hide Implementation Artifacts (SWE04)

Level → Advisory

Category

Safety
✓

Cyber
✓

Goal

Maintainability
✓

Reliability
✓

Portability
Performance
Security
✓

Remediation → High, as retrofit can be extensive

Verification Method → GNATcheck rule: `Visible_Components`

9.4.1 Reference

MISRA C Rule 8.7: "Functions and objects should not be defined with external linkage if they are referenced in only one translation unit."

9.4.2 Description

Do not make implementation artifacts compile-time visible to clients. Only make available those declarations that define the abstraction presented to clients by the component. In other words, define Abstract Data Types and use the language to enforce the abstraction. This is a fundamental Object-Oriented Design principle.

This guideline minimizes client dependencies and thus allows the maximum flexibility for changes in the underlying implementation. It minimizes the editing changes required for client code when implementation changes are made.

This guideline also limits the region of code required to find any bugs to the package and child packages, if any, defining the abstraction.

This guideline is to be followed extensively as the design default for components. Once the application code size becomes non-trivial, the cost of retrofit is extremely high.

9.4.3 Applicable Vulnerability within ISO TR 24772-2

N/A

9.4.4 Noncompliant Code Example

```
package Noncompliant is
   type Content_T is array (Capacity_T range <>) of Integer;
   type Stack_T (Capacity : Capacity_T) is tagged record
      Content : Content_T (1 .. Capacity);
      Top     : Capacity_T := 0;
   end record;
   procedure Push
     (Stack : in out Stack_T;
      Item  :        Integer);
   procedure Pop
     (Stack : in out Stack_T;
      Item  :    out Integer);
end Noncompliant;
```

Note that both type Content_T, as well as the record type components of type Stack_T, are visible to clients. Client code may declare variables of type Content_T and may directly access and modify the record components. Bugs introduced via this access could be anywhere in the entire client codebase.

9.4.5 Compliant Code Example

```
package Compliant is
   type Stack_T (Capacity : Capacity_T) is tagged private;
   procedure Push
     (Stack : in out Stack_T;
      Item  :        Integer);
   procedure Pop
     (Stack : in out Stack_T;
      Item  :    out Integer);
private
   type Content_T is array (Capacity_T range <>) of Integer;
   type Stack_T (Capacity : Capacity_T) is tagged record
      Content : Content_T (1 .. Capacity);
      Top     : Capacity_T := 0;
   end record;
end Compliant;
```

Type Content_T, as well as the record type components of type Stack_T, are no longer visible to clients. Any bugs in the stack processing code must be in this package, or its child packages, if any.

9.4.6 Notes

The GNATcheck rule specified above is not exhaustive.

CHAPTER

TEN

REFERENCES

- AdaCore. SPARK 2014 User's Guide.[20]
- Adacore. GNAT User's Guide for Native Platforms[21]
- AdaCore. "GNATstack User's Guide"[22]

[20] http://docs.adacore.com/spark2014-docs/html/ug/index.html
[21] http://docs.adacore.com/live/wave/gnat_ugn/html/gnat_ugn/gnat_ugn.html
[22] http://docs.adacore.com/live/wave/gnatstack/html/gnatstack_ug/index.html

BIBLIOGRAPHY

[SEI-C] The Software Engineering Institute. *SEI CERT C Coding Standard*.

[MISRA2013] MISRA. 2015. *Guidelines for the Use of the C Language in Critical Systems*

[Holzmann2006] Holzmann, G. J. 2006. *The Power of 10: Rules for Developing Safety-Critical Code*

[ISO2000] ISO/IEC High Integrity Rapporteur Group. 2000. "ISO/IEC TR 15942:2000 Guide for the Use of the Ada Programming Language in High Integrity Systems." *ISO/IEC TR 15942:2000*, July

[AdaRM2016] ISO/IEC. 2016. *ISO/IEC JTC 1/SC 22/WG9 Ada Reference Manual - Language and Standard Libraries-ISO/IEC 8652:2012/Cor 1:2016*

[AdaRM2020] ISO/IEC. 2020. *ISO/IEC JTC 1/SC 22/WG9 Ada Reference Manual - Language and Standard Libraries-ISO/IEC 8652:2020*

[AdaOOP2016] AdaCore. 2016. High-Integrity Object-Oriented Programming in Ada, Version 1.4[19]

[LiskovWing1994] Liskov, B. and Wing, J. 1994. "A Behavioral Notion of Subtyping." *ACM Transactions on Programming Languages and Systems (TOPLAS)* Vol. 16, Issue 6 (November): 1811-1841.

[DO178C] RTCA DO-178C/EUROCAE ED-12C. 2011. *Software Considerations in Airborne Systems and Equipment Certification*

[Meyer1997] Meyer, B. 1997. "Object-Oriented Software Construction." *Prentice Hall Professional Technical Reference* (2nd Edition)

[CWE2019] MITRE. 2019. *Common Weakness Enumeration (CWE)*

[SEI-Java] The Software Engineering Institute. *SEI CERT Oracle Coding Standard for Java*

[TR24772] ISO/IEC. 2022. *ISO/IEC TR 24772-2:20 Programming Languages - Guidance to Avoiding Vulnerabilities in Programming Languages - Part 2: Ada*

[19] https://www.adacore.com/uploads/techPapers/HighIntegrityAda.pdf