Guidelines for the development of safety-critical and security-critical applications using Ada and SPARK

[**Introduction**](#_2nakz7jgi2y0) **4**

[Scope](#_o9kv835n47h0) 4

[Structure](#_b0mrzjbi7vce) 4

[Enforcement](#_cdqx17381dg) 4

[**Guidelines**](#_v925vrha7mg2) **6**

[Dynamic Storage Management (DYN)](#_wo0xvt45quet) 6

[Realization](#_nejsaljjy2jq) 7

[Common High Integrity Restrictions (DYN01)](#_ucik0jm1ut2e) 7

[Traditional Static Allocation Policy (DYN02)](#_u2enoj4z3a8q) 8

[Access Types Without Allocators Policy (DYN03)](#_6fy1bb5ih2d1) 9

[Minimal Dynamic Allocation Policy (DYN04)](#_ad2i4usgbuxn) 10

[User-Defined Storage Pools Policy (DYN05)](#_crc8irbzr39b) 11

[Statically Determine Maximum Stack Requirements (DYN06)](#_dhjutna2e9r2) 12

[Safe Reclamation (RCL)](#_valietrq91fj) 13

[No Multiple Reclamations (RCL01)](#_2myhzjam25e1) 13

[Only Reclaim Allocated Storage (RCL02)](#_tcqfyg3u600i) 14

[Only Reclaim To The Same Pool (RCL03)](#_ltoj7aaa59dh) 15

[Concurrency (CON)](#_25k6phjgymu1) 16

[Use the Ravenscar Profile (CON01)](#_pqxeza4d4t7x) 17

[Use the Jorvik Profile (CON02)](#_4fpmxpgtfojb) 18

[Avoid Shared Variables for Inter-task Communication (CON03)](#_5wqg7rhov2c) 19

[Robust Programming Practice (RPP)](#_qwglk2dhgfbo) 20

[No Use of “others” in Case Constructs (RPP01)](#_q6345pkqqpn2) 20

[No Enumeration Ranges in Case Constructs (RPP02)](#_7ib1syy3nwx8) 21

[Limited Use of “others” In Aggregates (RPP03)](#_st7ersmkyjva) 22

[No Unassigned Mode-Out Procedure Parameters (RPP04)](#_sh9p6cixm0wt) 23

[No Use of “others” in Exception Handlers (RPP05)](#_pe7gcvka0z9m) 24

[Avoid Function Side-Effects (RPP06)](#_xvyopgaimoxi) 25

[Functions Only Have Mode “in” (RPP07)](#_kuwlthqndsbo) 26

[Limit Parameter Aliasing (RPP08)](#_dgojcjagquvz) 27

[Use Precondition and Postcondition Contracts (RPP09)](#_s2qimsm3kikn) 28

[Do Not Re-Verify Preconditions In Subprogram Bodies (RPP10)](#_s4wv3sy4q85r) 30

[Always Use the Result of Function Calls (RPP11)](#_6u191i9wql6l) 30

[No Recursion (RPP12)](#_i33u2n1zl48w) 31

[Exception Usage (EXU)](#_p9btlhckw1t7) 32

[Don’t Raise Language-Defined Exceptions (EXU01)](#_bd76sfo9shb2) 33

[No Unhandled Application-Defined Exceptions (EXU02)](#_5r0elusz6hbp) 34

[No Exception Propagation Beyond Name Visibility (EXU03)](#_d3xpjqlmin3b) 35

[Prove Absence of Run-time Exceptions (EXU04)](#_7r70pprkipz3) 36

[Object-Oriented Programming (OOP)](#_t7atpc7a1wmd) 37

[No Class-wide Constructs Policy (OOP01)](#_5guli2pivdh0) 38

[Static Dispatching Only Policy (OOP02)](#_xk7t69jeup90) 38

[Limit Inheritance Hierarchy Depth (OOP03)](#_6dyrp44ae7l) 39

[Limit Statically-Dispatched Calls To Primitive Operations (OOP04)](#_dcp0bgurnqoc) 40

[Use Explicit Overriding Annotations (OOP05)](#_dlfrohau25wm) 42

[Use Class-wide Pre/Post Contracts (OOP06)](#_8wc593ow1w01) 44

[Ensure Local Type Consistency (OOP07)](#_hmrapon82m43) 45

[Software Engineering (SWE)](#_lr9w4fw8ybhi) 50

[Use SPARK Extensively (SWE01)](#_gzt4hfuf40da) 50

[Enable Optional Warnings and Treat As Errors (SWE02)](#_hkzs38xsm6wx) 51

[Use A Static Analysis Tool Extensively (SWE03)](#_jq7f30idmako) 52

[Hide Implementation Artifacts (SWE04)](#_b5goesasad09) 53

[References](#_kvwv4gk0d911) 55

[**AdaCore Documentary License**](#_z5q205toa5vr) **57**

[Definition of the Material](#_3xo9i4cvz4qn) 57

[License](#_tbpmnxljrxoi) 57

[**GNU General Public License**](#_jino3r2bnb8k) **57**

# 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).

## 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.

## 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.

## 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 [[5](#_tyjcwt)]. 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.

# Guidelines

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 will have to think about which guidelines to apply and how best to apply each guideline selected.

## Dynamic Storage Management (DYN)

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| DYN | Dynamic Storage Management | | | | | | | | | |
| Goal | X | Maintainability | X | Reliability |  | portability | X | performance | X | 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 [[1-4](#_30j0zll)].

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.

Applicable vulnerability within ISO TR 24772-2:

* 6.39 "Memory leak and heap fragmentation [XYL].

#### 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.

### Common High Integrity Restrictions (DYN01)

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| DYN01 | | | | | Safety | | X | Cyber | X | Required |
| Common High Integrity Restrictions | | | | | | | | | | |
| Goal | X | Maintainability | X | Reliability |  | Portability |  | Performance | X | Security |
| 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 un-reclaimed (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.) | | | | | | | | | |
| Reference | Ada RM, High Integrity Annex Restrictions, section H.4 | | | | | | Remediation | | Low | |
| 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")); | | | | | | | | | |
| 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); | | | | | | | | | |

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.

### Traditional Static Allocation Policy (DYN02)

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| DYN02 | | | | | Safety | | X | Cyber | | X | Advisory |
| Traditional Static Allocation Policy | | | | | | | | | | | |
| Goal | X | Maintainability | X | Reliability |  | Portability |  | Performance | | X | Security |
| 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 | | | | | | | | | | |
| Reference | MISRA C Dir 4.12 "Dynamic memory allocation shall not be used" | | | | | | Remediation | | Low | | |
| Noncompliant Code Example | Any code using the constructs listed above. | | | | | | | | | | |
| Compliant Code Example | N/A | | | | | | | | | | |

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

### Access Types Without Allocators Policy (DYN03)

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| DYN03 | | | | | | Safety | X | Cyber | | | X | Advisory |
| Access Types Without Allocators Policy | | | | | | | | | | | | |
| Goal | X | Maintainability | X | Reliability |  | Portability |  | | Performance | | X | Security |
| 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   **Notes**  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 RM 13.11.3(6.1/3)). | | | | | | | | | | | |
| Reference | MISRA rule 21.3 "The memory allocation and deallocation functions of <stdlib.h> shall not be used" | | | | | | Remediation | | | Low | | |
| Noncompliant Code Example | Any code using the constructs listed above. | | | | | | | | | | | |
| Compliant Code Example | **type** Descriptor is …;  **type** Descriptor\_Ref **is access all** Descriptor;  …  Device : **aliased** Descriptor;  …  P : Descriptor\_Ref := Device’Access;  … | | | | | | | | | | | |

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

### Minimal Dynamic Allocation Policy (DYN04)

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| DYN04 | | | | | Safety | | X | | Cyber | X | Advisory |
| Minimal Dynamic Allocation Policy | | | | | | | | | | | |
| Goal | X | Maintainability | X | Reliability |  | Portability |  | Performance | | X | Security |
| 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. | | | | | | | | | | |
| Reference | Power of Ten rule 3 "Do not use dynamic memory allocation after initialization" | | | | | | Remediation | | | Low | |
| Noncompliant Code Example | Any code using the constructs listed above. | | | | | | | | | | |
| Compliant Code Example | Code performing dynamic allocations any time prior to an arbitrary point designated as the end of the “startup” interval. | | | | | | | | | | |

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

### User-Defined Storage Pools Policy (DYN05)

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| DYN05 | | | | | | Safety | X | Cyber | X | Advisory |
| User-Defined Storage Pools | | | | | | | | | | |
| Goal | X | Maintainability | X | Reliability |  | Portability |  | Performance | X | Security |
| 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. | | | | | | | | | |
| Reference | MISRA rule 21.3 "The memory allocation and deallocation functions of <stdlib.h> shall not be used" | | | | | | Remediation | | Low | |
| Noncompliant Code Example | Allocation via an access type not tied to a user-defined storage pool. | | | | | | | | | |
| 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 | | | | | | | | | |

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

### Statically Determine Maximum Stack Requirements (DYN06)

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| DYN06 | | | | | Safety | | X | | Cyber | X | Required |
| Statically Determine Maximum Stack Requirements Per Task | | | | | | | | | | | |
| Goal | X | Maintainability | X | Reliability |  | Portability |  | Performance | | X | Security |
| 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.) | | | | | | | | | | |
| Reference | N/A | | | | | | Remediation | | | High | |
| Noncompliant Code Example | N/A | | | | | | | | | | |
| Compliant Code Example | N/A | | | | | | | | | | |

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

## Safe Reclamation (RCL)

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| RCL | Safe Reclamation | | | | | | | | | |
| Goal | X | Maintainability | X | Reliability |  | portability | X | performance | X | 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 | | | | | | | | | |

### No Multiple Reclamations (RCL01)

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| RCL01 | | | | | | Safety | X | Cyber | X | Mandatory |
| No Multiple Reclamations | | | | | | | | | | |
| Goal | X | Maintainability | X | Reliability | X | Portability |  | Performance | X | Security |
| Description | Never deallocate the storage designated by a given access value more than once. | | | | | | | | | |
| Reference | [14] CWE-415: Double Free | | | | | | Remediation | | High | |
| 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); | | | | | | | | | |
| Compliant Code Example | Remove the call to Free (Y). | | | | | | | | | |

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.

### Only Reclaim Allocated Storage (RCL02)

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| RCL02 | | | | | | Safety | X | Cyber | X | Mandatory |
| Only Reclaim Allocated Storage | | | | | | | | | | |
| Goal | X | Maintainability | X | Reliability | X | Portability |  | Performance | X | Security |
| Description | Only deallocate storage that was dynamically allocated by the evaluation of an allocator (i.e., “new”).  This is a possibility because Ada allows creation of access values designating declared (aliased) objects. | | | | | | | | | |
| Reference | [1] MEM34-C: Only Free Memory Allocated Dynamically | | | | | | Remediation | | High | |
| 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); | | | | | | | | | |
| Compliant Code Example | Remove the call to Free (Y). | | | | | | | | | |

Enforcement of this rule can only be provided by manual code review, unless deallocation is forbidden via No\_Unchecked\_Deallocation.

### Only Reclaim To The Same Pool (RCL03)

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| RCL03 | | | | | | Safety | X | Cyber | X | Mandatory |
| Only Reclaim Allocated Storage Into The Storage Pool From Which It Was Allocated | | | | | | | | | | |
| Goal | X | Maintainability | X | Reliability | X | Portability |  | Performance | X | Security |
| Description | When deallocating, ensure that the pool to which the storage will be returned was the same pool from which it was allocated. Execution is erroneous otherwise, meaning anything can happen (RM 13.11.2(16)).  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. | | | | | | | | | |
| Reference | N/A | | | | | | Remediation | | High | |
| 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);  ...  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. | | | | | | | | | |
| Compliant Code Example | Don’t deallocate converted access values. | | | | | | | | | |

Enforcement of this rule can only be provided by manual code review, unless deallocation is forbidden via No\_Unchecked\_Deallocation.

## Concurrency (CON)

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| CON | Concurrency | | | | | | | | | |
| Goal | X | Maintainability | X | Reliability | X | Portability | X | Performance |  | Security |
| Description | Have a plan for managing the use of concurrency in high-integrity applications having real-time requirements. | | | | | | | | | |
| Rules | CON01, CON02, CON02 | | | | | | | | | |

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 [5] 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 [6] 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.

### Use the Ravenscar Profile (CON01)

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| CON01 | | | | | Safety | | X | | Cyber |  | Advised |
| Apply the Ravenscar Profile | | | | | | | | | | | |
| Goal | X | Maintainability | X | Reliability | X | Portability | X | Performance | |  | Security |
| 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); | | | | | | | | | | |
| Reference | Ada RM section D.13 | | | | | | Remediation | | | High | |
| 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. | | | | | | | | | | |
| Compliant Code Example | N/A | | | | | | | | | | |

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

Applicable vulnerability within ISO TR 24772-2:

* 6.63 "Lock protocol errors [CGM]".

### Use the Jorvik Profile (CON02)

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| CON02 | | | | | Safety | | X | | Cyber | X | Advised |
| Apply the Jorvik Profile | | | | | | | | | | | |
| Goal | X | Maintainability | X | Reliability | X | Portability | X | Performance | |  | Security |
| 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);  These restrictions are removed from Ravenscar:  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 Simple\_Barriers). | | | | | | | | | | |
| Reference | Ada 202x RM section D.13 | | | | | | Remediation | | | High | |
| 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. | | | | | | | | | | |
| Compliant Code Example | N/A | | | | | | | | | | |

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

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

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| CON03 | | | | | Safety | | X | | Cyber | X | Advised |
| Avoid shared variables for communication between tasks | | | | | | | | | | | |
| Goal | X | Maintainability | X | Reliability | X | Portability | X | Performance | |  | Security |
| 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. | | | | | | | | | | |
| Reference | Ada RM section D.13 | | | | | | Remediation | | | Medium | |
| Noncompliant Code Example | A variable marked as Volatile but not assigned to a specific address in memory:  X : Integer **with** Volatile;  Note that variables marked as Atomic are also Volatile, per the Ada RM C.6/8(3). | | | | | | | | | | |
| 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.  GPIO\_A : GPIO\_Port  **with** Import, Volatile, Address => GPIOA\_Base; | | | | | | | | | | |

GNATcheck can detect violations via the Volatile\_Objects\_Without\_Address\_Clauses rule. SPARK and CodePeer can also detect conflicting access to unprotected variables.

Applicable vulnerability within ISO TR 24772-2:

* 6.56 "Undefined behaviour [EWF]".

## Robust Programming Practice (RPP)

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| RPP | Robust Programming Practice | | | | | | | | | |
| Goal | X | Maintainability | X | Reliability | X | Portability | X | Performance | X | Security |
| Description | These rules promote the production of robust software. | | | | | | | | | |
| Rules | RPP01, RPP02, RPP03, RPP04, RPP05, RPP06, RPP07, RPP07, RPP08, RPP09, RPP10, RPP11, RPP12 | | | | | | | | | |

### No Use of “others” in Case Constructs (RPP01)

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| RPP01 | | | | | Safety | | X | | Cyber | X | Required |
| No Use of “others” in Case Constructs | | | | | | | | | | | |
| Goal | X | Maintainability | X | Reliability | X | Portability |  | Performance | |  | Security |
| 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 [1] MSC01-C. The reason is that in C, “default” alternative plays the role of defensive code to mitigate the switch statement’s non-exhaustivity. In Ada, the case construct is exhaustive: 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. | | | | | | | | | | |
| Reference | [1] MSC01-C | | | | | | Remediation | | | Low | |
| Noncompliant Code Example | **type** Agency **is** ( ESA, NASA, RFSA, JAXA, CNSA);  -- there are dozens...  Bureau : Agency;  …  **case** Bureau **is**  …  **when** others => …  **end case**; | | | | | | | | | | |
| Compliant Code Example | **type** Agency **is** ( ESA, NASA, RFSA, JAXA, CNSA);  -- there are dozens...  Bureau : Agency;  …  **case** Bureau **is**  **when** ESA => ...  **when** NASA => ...  **when** RFSA => ...  **when** JAXA => ...  **when** CNSA => ...  **end case**; | | | | | | | | | | |

GNATcheck can detect violations via the OTHERS\_In\_CASE\_Statementsrule.

Applicable vulnerability within ISO TR 24772-2:

* 6.39.1 "Unanticipated exceptions from library routines [HJW]".

### No Enumeration Ranges in Case Constructs (RPP02)

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| RPP02 | | | | | Safety | | X | | Cyber | X | Required |
| No Enumeration Ranges in Case Constructs | | | | | | | | | | | |
| Goal | X | Maintainability | X | Reliability | X | Portability |  | Performance | |  | Security |
| 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. | | | | | | | | | | |
| Reference | Similar to RPP01 | | | | | | Remediation | | | Low | |
| Noncompliant Code Example | **type** Agency is (ESA, NASA, RFSA, JAXA, CNSA);  -- there are dozens...  Bureau : Agency;  ...  **case** Bureau **is**  **when** ESA .. RFSA => Do\_Something;  **when** ...  **end case**; | | | | | | | | | | |
| Compliant Code Example | **type** Agency **is** (ESA, NASA, RFSA, JAXA, CNSA);  -- there are dozens...  Bureau : Agency;  ...  **case** Bureau **is**  **when** ESA | NASA | RFSA => Do\_Something  **when** ...  **end case**; | | | | | | | | | | |

GNATcheck can detect violations via the Enumeration\_Ranges\_In\_CASE\_Statementsrule.

Applicable vulnerability within ISO TR 24772-2:

* 6.5 "Enumerator issues [CCB]".

### Limited Use of “others” In Aggregates (RPP03)

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| RPP03 | | | | | Safety | | X | | Cyber | X | Advised |
| Limited Use of “others” In Aggregates | | | | | | | | | | | |
| Goal | X | Maintainability | X | Reliability | X | Portability |  | Performance | |  | Security |
| 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. | | | | | | | | | | |
| Reference | Similar to RPP01 | | | | | | Remediation | | | Low | |
| Noncompliant Code Example | **type** Agency **is** (ESA, NASA, RFSA, JAXA, CNSA);  -- there are dozens...    **type** Agencies\_Mask **is array** (Agency) **of** Boolean;    Partners : Agencies\_Mask := (NASA | ESA | JAXA | RFSA => True, **others** => False); | | | | | | | | | | |
| Compliant Code Example | Partners : **constant** Agencies\_Mask := (CNSA => False, others => True);  In this example, the “others” is allowed because it refers to all but one component. | | | | | | | | | | |

GNATcheck can detect violations via the OTHERS\_In\_Aggregatesrule.

### No Unassigned Mode-Out Procedure Parameters (RPP04)

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| RPP04 | | | | | Safety | | X | | Cyber | X | Required |
| No Unassigned Mode-Out Procedure Parameters | | | | | | | | | | | |
| Goal | X | Maintainability | X | Reliability | X | Portability |  | Performance | |  | Security |
| 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.  Remediation is High because of the difficulty of finding these bugs by manual inspection. | | | | | | | | | | |
| Reference | MISRA C rule 9.1 "The value of an object with automatic storage duration shall not be read before it has been set" | | | | | | Remediation | | | High | |
| Noncompliant Code Example | **type** Agency **is** (ESA, NASA, RFSA, JAXA, CNSA);  -- there are dozens...  **for** Agency **use**  (ESA => 1, NASA => 3, RFSA => 5, JAXA => 7, CNSA => 9);  Bureau : Agency := RFSA;    **procedure** Update  (Input : **in** Boolean;  Partner : **out** Agency)  **is**  **begin**  **if** Input **then**  Partner := ...  **end if**;  **end** Update;  In the above, some value is copied back for the second formal parameter Partner, but the value is only defined if the first parameter is True. That 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.) | | | | | | | | | | |
| Compliant Code Example | **type** Agency **is** (ESA, NASA, RFSA, JAXA, CNSA);  **-- there are dozens...**  **for** Agency **use**  (ESA => 1, NASA => 3, RFSA => 5, JAXA => 7, CNSA => 9);  Bureau : Agency := RFSA;    **procedure** Update  (Input : **in** Boolean;  Partner : **out** Agency)  **is**  **begin**  **if** Input **then**  Partner := ...  **else**  Partner := ...  **end if**;  **end** Update; | | | | | | | | | | |

GNATcheck can detect violations via the Unassigned\_OUT\_Parametersrule.

**Warning**: This rule 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.

Applicable vulnerability within ISO TR 24772-2:

* 6.32 "Passing parameters and return values [CSJ]".

### No Use of “others” in Exception Handlers (RPP05)

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| RPP05 | | | | | Safety | | X | | Cyber | X | Required |
| No Use of “others” in Exception Handlers | | | | | | | | | | | |
| Goal | X | Maintainability | X | Reliability | X | Portability |  | Performance | |  | Security |
| 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. | | | | | | | | | | |
| Reference | N/A | | | | | | Remediation | | | Low | |
| Noncompliant Code Example | **exception**  **when others** =>  ... | | | | | | | | | | |
| Compliant Code Example | Code that references all handled exceptions by their names. | | | | | | | | | | |

GNATcheck can detect violations via the OTHERS\_In\_Exception\_Handlersrule.

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 =>.

It also recommends "All tasks should contain an exception handler at the outer level to prevent silent termination due to unhandled exceptions." for vulnerability 6.62 Concurrency - Premature termination.

### 

### Avoid Function Side-Effects (RPP06)

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| RPP06 | | | | | Safety | | X | | Cyber | X | Advised |
| Avoid Function Side-Effects | | | | | | | | | | | |
| Goal | X | Maintainability | X | Reliability | X | Portability |  | Performance | |  | Security |
| 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. | | | | | | | | | | |
| Reference | MISRA C rule 13.3 "The value of an expression and its persistent side effects shall be the same under all permitted evaluation orders" | | | | | | Remediation | | | Medium | |
| Noncompliant Code Example | Call\_Count : Integer := 0;    **function** F **return** Boolean **is**  Result : Boolean;  **begin**  ...  Call\_Count := Call\_Count + 1;  **return** Result;  **end** F; | | | | | | | | | | |
| Compliant Code Example | Remove the update to Call\_Count. or change the function into a procedure with a parameter for Call\_Count. | | | | | | | | | | |

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

Applicable vulnerability within ISO TR 24772-2:

* 6.24 "Side-effects and order of evaluation [SAM]".

### Functions Only Have Mode “in” (RPP07)

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| RPP07 | | | | | Safety | | X | | Cyber | X | Required |
| Functions Only Have Mode “in” | | | | | | | | | | | |
| Goal | X | Maintainability | X | Reliability | X | Portability |  | Performance | |  | Security |
| 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. | | | | | | | | | | |
| Reference | RP07 | | | | | | Remediation | | | Low | |
| Noncompliant Code Example | **function** Square (Input : in out Integer) **return** Integer **is**  Result : Integer;  **begin**  Result := Input \* Input;  Input := Input + 1;  **return** Result;  **end** Square; | | | | | | | | | | |
| Compliant Code Example | **function** Square (Input : in Integer) **return** Integer **is**  Result : Integer;  **begin**  Result := Input \* Input;  **return** Result;  **end** Square;  or  **function** Square (Input : in Integer) **return** Integer **is**  (Input \* Input); | | | | | | | | | | |

Violations are detected by SPARK.

Applicable vulnerability within ISO TR 24772-2:

* 6.24 "Side-effects and order of evaluation [SAM]".

### Limit Parameter Aliasing (RPP08)

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| RPP08 | | | | | Safety | | X | | Cyber | X | Required |
| Limit Parameter Aliasing | | | | | | | | | | | |
| Goal | X | Maintainability | X | Reliability | X | Portability |  | Performance | |  | Security |
| 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.  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. | | | | | | | | | | |
| Reference | Ada RM section 6.2  SPARK RM section 6.4.2 | | | | | | Remediation | | | High | |
| 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); | | | | | | | | | | |
| Compliant Code Example | Don’t pass Obj as the actual parameter to both formal parameters. | | | | | | | | | | |

All violations are detected by SPARK. The GNAT compiler switch “-gnateA” enables detection of some cases, but not all.

### Use Precondition and Postcondition Contracts (RPP09)

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| RPP09 | | | | | Safety | | X | | Cyber | X | Advised |
| Use Precondition and Postcondition Contracts | | | | | | | | | | | |
| Goal | X | Maintainability | X | Reliability | X | Portability |  | Performance | | X | Security |
| 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. | | | | | | | | | | |
| Reference | Power of Ten rule 5 "The assertion density of the code should average to a minimum of two assertions per function." | | | | | | Remediation | | | Low | |
| Noncompliant Code Example | **type** Stack **is private**;  **procedure** Push (This : **in out** Stack; Item : Element); | | | | | | | | | | |
| 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**; | | | | | | | | | | |

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.

Applicable vulnerability within ISO TR 24772-2:

* 6.42 "Violations of the Liskov substitution principle or the contract model [BLP]".

### Do Not Re-Verify Preconditions In Subprogram Bodies (RPP10)

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| RPP10 | | | | | Safety | | X | | Cyber |  | Required |
| Do Not Verify Preconditions In Subprogram Bodies | | | | | | | | | | | |
| Goal | X | Maintainability | X | Reliability | X | Portability |  | Performance | |  | Security |
| Description | Do not re-verify preconditions in the corresponding subprogram bodies. It is a waste of cycles and confuses the reader as well. | | | | | | | | | | |
| Reference | RPP10 | | | | | | Remediation | | | Low | |
| 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; | | | | | | | | | | |
| 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; | | | | | | | | | | |

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

### Always Use the Result of Function Calls (RPP11)

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| RPP11 | | | | | Safety | | X | | Cyber | X | Advised |
| Always Use the Results of Function Calls | | | | | | | | | | | |
| Goal | X | Maintainability | X | Reliability | X | Portability |  | Performance | |  | Security |
| 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. | | | | | | | | | | |
| 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." | | | | | | Remediation | | | Low | |
| Noncompliant Code Example | N/A | | | | | | | | | | |
| Compliant Code Example | N/A | | | | | | | | | | |

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.

Applicable vulnerability within ISO TR 24772-2:

* 6.47 "Inter-language calling [DJS]"

### No Recursion (RPP12)

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| RPP12 | | | | | Safety | | X | | Cyber | X | Advised |
| No Recursive Subprograms | | | | | | | | | | | |
| Goal | X | Maintainability | X | Reliability | X | Portability |  | Performance | |  | Security |
| 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 manual supply of call limits (for example). | | | | | | | | | | |
| Reference | MISRA C rule 17.2 "Functions shall not call themselves, either directly or indirectly" | | | | | | Remediation | | | Low | |
| Noncompliant Code Example | **function** Factorial (N : Positive) **return** Positive **is**  **begin**  **if** N = 1 **then**  **return** 1;  **else**  **return** N \* Factorial (N - 1); -- could overflow  **end if**;  **end** Factorial; | | | | | | | | | | |
| Compliant Code Example | **function** Factorial (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** Factorial; | | | | | | | | | | |

The compiler will detect violations with the restriction No\_Recursion in place. Note this is a dynamic check. GNATcheck enforces it statically with +RRecursive\_Subprograms, subject to the limitations described in<http://docs.adacore.com/live/wave/asis/html/gnatcheck_rm/gnatcheck_rm/predefined_rules.html#recursive-subprograms>.

Applicable vulnerability within ISO TR 24772-2:

* 6.35 "Recursion [GDL]"

## Exception Usage (EXU)

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| EXU | Exception Usage | | | | | | | | | |
| Goal | X | Maintainability | X | Reliability | X | Portability | X | Performance | X | 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.

Applicable vulnerability within ISO TR 24772-2:

* 6.36 "Ignored error status and unhandled exceptions [OYB]"
* 6.50 "Unanticipated exceptions from library routines [HJW]"

### Don’t Raise Language-Defined Exceptions (EXU01)

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| EXU01 | | | | | Safety | | X | | Cyber | X | Required |
| Don’t Raise Language-Defined Exceptions | | | | | | | | | | | |
| Goal | X | Maintainability | X | Reliability | X | Portability |  | Performance | |  | Security |
| 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. | | | | | | | | | | |
| Reference | [15] ERR07-J | | | | | | Remediation | | | Low | |
| Noncompliant Code Example | **if** Fuel\_Exhaused (This) **then**  **raise** Constraint\_Error;  **end if**; | | | | | | | | | | |
| Compliant Code Example | Fuel\_Limits : exception;  ...  **if** Fuel\_Exhaused (This) **then**  **raise** Fuel\_Limits;  **end if**; | | | | | | | | | | |

This restriction is detected by GNATcheck with the Raising\_Predefined\_Exceptions rule applied.

### No Unhandled Application-Defined Exceptions (EXU02)

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| EXU02 | | | | | Safety | | X | | Cyber | X | Required |
| No Unhandled Application-defined Exceptions | | | | | | | | | | | |
| Goal | X | Maintainability | X | Reliability | X | Portability |  | Performance | |  | Security |
| 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 occurrence is raised in the sequence of statements of an application task and reaches the body of that task, the task completes 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 RM 10.2(30). 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 (RM 10.2(34). In some systems there is no calling environment, such as bare-metal platforms with only an Ada run-time library and no operating system. | | | | | | | | | | |
| Reference | N/A | | | | | | Remediation | | | Low | |
| Noncompliant Code Example | N/A | | | | | | | | | | |
| Compliant Code Example | N/A | | | | | | | | | | |

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

### No Exception Propagation Beyond Name Visibility (EXU03)

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| EXU03 | | | | | Safety | | X | | Cyber | X | Required |
| No Exception Propagation Beyond Name Visibility | | | | | | | | | | | |
| Goal | X | Maintainability | X | Reliability | X | Portability |  | Performance | |  | Security |
| 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. | | | | | | | | | | |
| Reference | RPP05 | | | | | | Remediation | | | Low | |
| Noncompliant Code Example | **package** P **is**  **procedure** Q;  **end** P;    **package body** P **is**    Error : **exception**;    **procedure** Q **is**  **begin**  ...  **raise** Error; -- under some circumstance  ...  **end** Q;    **end** P;  As a result the exception name cannot be referenced outside the body:  **begin** -- some code outside of P  P.Q;  **exception**  **when** P.Error => -- illegal | | | | | | | | | | |
| Compliant Code Example | Either make the exception name visible to clients:  **package** P **is**  Error : exception; -- moved from package body  **procedure** Q;  **end** P;  or ensure the exception is not propagated beyond the scope of its declaration:  **package body** P **is**    Error : **exception**;    **procedure** Q **is**  **begin**  ...  **raise** Error; -- under some circumstance  ...  **exception**  **when** Error => ...  **end** Q;    **end** P; | | | | | | | | | | |

GNATcheck can detect violations via the Non\_Visible\_Exceptionsrule.

### Prove Absence of Run-time Exceptions (EXU04)

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| EXU04 | | | | | Safety | | X | | Cyber | X | Required |
| Prove Absence of Run-time Exceptions | | | | | | | | | | | |
| Goal | X | Maintainability | X | Reliability | X | Portability |  | Performance | |  | Security |
| 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. | | | | | | | | | | |
| Reference | MISRA C rule 1.3 "There shall be no occurrence of undefined or critical unspecified behaviour" | | | | | | Remediation | | | High | |
| Noncompliant Code Example | N/A | | | | | | | | | | |
| Compliant Code Example | N/A | | | | | | | | | | |

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.

## Object-Oriented Programming (OOP)

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| OOP | Object-Oriented Programming | | | | | | | | | |
| Goal | X | Maintainability | X | Reliability |  | portability |  | performance | X | 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.

### No Class-wide Constructs Policy (OOP01)

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| OOP01 | | | | | Safety | | X | Cyber | X | Advised |
| No Class-wide Constructs Policy | | | | | | | | | | |
| Goal | X | Maintainability | X | Reliability |  | Portability |  | Performance | X | Security |
| 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. | | | | | | | | | |
| Reference | N/A | | | | | | Remediation | |  | |
| Noncompliant Code Example | X : Object'Class := Some\_Object; | | | | | | | | | |
| Compliant Code Example | X : Object := Some\_Object; | | | | | | | | | |

The compiler will detect violations with the standard Ada restriction No\_Dispatch applied.

Applicable vulnerability within ISO TR 24772-2:

* 6.43 "Redispatching [PPH]"

### Static Dispatching Only Policy (OOP02)

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| OOP02 | | | | | Safety | | X | Cyber | X | Advised |
| Static Dispatching Only | | | | | | | | | | |
| Goal | X | Maintainability | X | Reliability |  | Portability |  | Performance | X | Security |
| 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. | | | | | | | | | |
| Reference | N/A | | | | | | Remediation | |  | |
| Noncompliant Code Example | Some\_Primitive (Object'Class (X)); | | | | | | | | | |
| Compliant Code Example | Some\_Primitive (X); | | | | | | | | | |

The compiler will detect violations with the GNAT-defined restriction No\_Dispatching\_Calls applied.

Applicable vulnerability within ISO TR 24772-2:

* 6.43 "Redispatching [PPH]"

### Limit Inheritance Hierarchy Depth (OOP03)

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| OOP03 | | | | | Safety | | X | Cyber | X | Advised |
| Limit Inheritance Hierarchy Depth | | | | | | | | | | |
| Goal | X | Maintainability | X | Reliability |  | Portability |  | Performance | X | Security |
| 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. | | | | | | | | | |
| Reference | [7] section 5.1 | | | | | | Remediation | | High | |
| Noncompliant Code Example | The threshold for “too deep” is inexact, but beyond around 4 or 5 levels the complexity accelerates rapidly. | | | | | | | | | |
| Compliant Code Example | N/A | | | | | | | | | |

Violations can be detected with the GNATcheck tool parameter Deep\_Inheritance\_Hierarchies, specifying a maximum inheritance depth as a parameter of the rule.

Applicable vulnerability within ISO TR 24772-2:

* 6.41 "Inheritance [RIP]"

### Limit Statically-Dispatched Calls To Primitive Operations (OOP04)

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| OOP04 | | | | | Safety | | X | | Cyber | X | Advised |
| Limit Statically-Dispatched Calls To Primitive Operations of the Same Tagged Type | | | | | | | | | | | |
| Goal | X | Maintainability | X | Reliability | X | Portability |  | Performance | |  | Security |
| 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”.) | | | | | | | | | | |
| Reference |  | | | | | | Remediation | | | Medium (easy fix, but a difficult to detect bug) | |
| Noncompliant Code Example | **package** Graphics **is**  **type** Shape **is tagged** -- really, abstract and private  **record**  X : Float := 0.0;  Y : Float := 0.0;  **end record**;  **function** Area (This : Shape) **return** Float;  -- would really be abstract  **function** Momentum (This : Shape) **return** Float;  ...  **end** Graphics;  **package body** Graphics **is**  **function** Area (This : Shape) **return** Float **is**  (0.0);  **function** Momentum (This : Shape) **return** Float **is**  **begin**  **return** This.X \* Area (This); -- wrong, but legal  **end** Momentum;  ...  **end** Graphics;  In the (somewhat artificial) example above, Momentum always returns zero because it always calls the Area function for type Shape. | | | | | | | | | | |
| Compliant Code Example | **package body** Graphics **is**  ...  **function** Momentum (This : Shape) **return** Float **is**  **begin**  **return** This.X \* Area (Shape'Class (This));  -- redispatch to an overriding for Area, if any  **end** Momentum;  ...  **end** Graphics; | | | | | | | | | | |

This rule can be enforced by GNATcheck with the Direct\_Calls\_To\_Primitives rule applied. The rule parameter Except\_Constructors may be added for constructor functions.

### Use Explicit Overriding Annotations (OOP05)

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| OOP05 | | | | | Safety | | X | | Cyber | X | Required |
| Use Explicit Overriding Annotations | | | | | | | | | | | |
| Goal | X | Maintainability | X | Reliability | X | Portability |  | Performance | |  | Security |
| 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. | | | | | | | | | | |
| Reference | [7] Section 4.3 | | | | | | Remediation | | | Low | |
| Noncompliant Code Example | **type** Generator **is new** Ada.Finalization.Controlled **with** …  -- really overriding, but not marked as such  **procedure** Initialize (This : **in out** Generator);  **overriding** -- marked but not really overriding  **procedure** Initialise (This : **in out** Generator); | | | | | | | | | | |
| Compliant Code Example | **type** Generator **is new** Ada.Finalization.Controlled **with** …  **overriding**  **procedure** Initialize (This : **in out** Generator);  **procedure** Initialise (This : **in out** Generator); | | | | | | | | | | |

This rule requires the GNAT compiler switches “-gnatyO” 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. Alternatively, GNATcheck will flag those errors via the “+Style\_Checks:O” rule.

Applicable vulnerability within ISO TR 24772-2:

* 6.41 "Inheritance [RIP]"

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

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| OOP06 | | | | | Safety | | X | Cyber | X | Required |
| Use Class-wide Pre/Post Contracts | | | | | | | | | | |
| Goal | X | Maintainability | X | Reliability |  | Portability |  | Performance | X | Security |
| 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). | | | | | | | | | |
| Reference | [7] Section 6.1.4; [8] section 7.5.2 | | | | | | Remediation | | Low | |
| Noncompliant Code Example | **type** Stack **is tagged** …  **function** Top\_Element (This : Stack) **return** Element **with**  Pre => **not** Empty (This),  ... | | | | | | | | | |
| Compliant Code Example | **type** Stack **is tagged** …  **function** Top\_Element (This : Stack) **return** Element **with**  Pre'Class => **not** Empty (This),  ... | | | | | | | | | |

Violations can be detected with the GNATcheck rule Specific\_Pre\_Post. SPARK enforces this guideline automatically.

Applicable vulnerability within ISO TR 24772-2:

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

### Ensure Local Type Consistency (OOP07)

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| OOP07 | | | | | Safety | | X | Cyber | X | Required |
| Ensure Local Type Consistency | | | | | | | | | | |
| Goal | X | Maintainability | X | Reliability |  | Portability |  | Performance | X | Security |
| 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.  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 [10] 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 [11] 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 [12]. 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.  For further discussion of this topic, see the sections cited in the Reference entry in this table. | | | | | | | | | |
| Reference | [7] See section 4.2.  [9] See section 5.10.11. | | | | | | Remediation | | High (the correction is syntactically trivial individually, but would be required throughout the tree). | |
| 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)  **with**  Post => Width (This) = Height (This);  **private**  …  **end** Q; | | | | | | | | | |
| 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; | | | | | | | | | |

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

## Software Engineering (SWE)

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| SWE | Software Engineering | | | | | | | | | |
| Goal | X | Maintainability | X | Reliability | X | Portability |  | Performance | X | Security |
| Description | These rules promote “best practices” for software development. | | | | | | | | | |
| Rules | SWE01, SWE02, SWE03, SWEP04 | | | | | | | | | |

### Use SPARK Extensively (SWE01)

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| SWE01 | | | | | Safety | | X | Cyber | X | Advised |
| Use SPARK Extensively | | | | | | | | | | |
| Goal | X | Maintainability | X | Reliability | X | Portability | X | Performance | X | Security |
| Description | SPARK has proven itself highly effective, both in terms of low defects, low development costs, and high productivity. The guideline advises extensive 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. | | | | | | | | | |
| Reference | [8] Section 8 “Applying SPARK in Practice” | | | | | | Remediation | | High, as retrofit can be extensive | |
| Noncompliant Code Example | Any code outside the (very large) SPARK subset is flagged by the compiler. | | | | | | | | | |
| Compliant Code Example | N/A | | | | | | | | | |

Violations are detected by the SPARK toolset.

### Enable Optional Warnings and Treat As Errors (SWE02)

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| SWE02 | | | | | Safety | | X | Cyber | X | Required |
| Enable Options Warnings and Treat As Errors | | | | | | | | | | |
| Goal | X | Maintainability | X | Reliability |  | Portability |  | Performance | X | Security |
| 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. | | | | | | | | | |
| 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. | | | | | | Remediation | | Low | |
| 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. | | | | | | | | | |
| 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. | | | | | | | | | |

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.

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]".

### Use A Static Analysis Tool Extensively (SWE03)

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| SWE03 | | | | | Safety | | X | Cyber | X | Mandatory |
| Use A Static Analysis Extensively | | | | | | | | | | |
| Goal | X | Maintainability | X | Reliability | X | Portability | X | Performance | X | Security |
| 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. | | | | | | | | | |
| 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. | | | | | | Remediation | | High | |
| Noncompliant Code Example | N/A | | | | | | | | | |
| Compliant Code Example | N/A | | | | | | | | | |

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

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]"

### Hide Implementation Artifacts (SWE04)

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| SWE04 | | | | | Safety | | X | Cyber | X | Advised |
| Hide Implementation Artifacts | | | | | | | | | | |
| Goal | X | Maintainability | X | Reliability |  | Portability |  | Performance | X | Security |
| 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. | | | | | | | | | |
| 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" | | | | | | Remediation | | High | |
| Noncompliant Code Example | **generic**  …  **package** Bounded\_Stacks **is**  **type** Content **is**  **array** (Physical\_Capacity **range** <>) **of** Element;  **type** Stack (Capacity : Physical\_Capacity) **is**  **tagged record**  Values : Content (1 .. Capacity);  Top : Element\_Count := 0;  **end record**;  **procedure** Push (This : in out Stack; …  -- additional primitives …  **end** Bounded\_Stacks;  Note that both type Content, as well as the record type components of type Stack, are visible to clients. Client code may declare variables of type Content and may directly access and modify the record components. Bugs introduced via this access could be anywhere in the entire client codebase. | | | | | | | | | |
| Compliant Code Example | **generic**  …  **package** Bounded\_Stacks **is**  **type** Stack (Capacity : Physical\_Capacity) **is**  **tagged private**;  **procedure** Push (This : **in out** Stack; …  -- additional primitives …  **private**  **type** Content **is**  **array** (Physical\_Capacity **range** <>) **of** Element;  **type** Stack (Capacity : Physical\_Capacity) **is**  **tagged record**  Values : Content (1 .. Capacity);  Top : Element\_Count := 0;  **end record**;  **end** Bounded\_Stacks;  Type Content, as well as the record type components of type Stack, are no longer visible to clients. Any bugs in the Stack code ***must*** be in this package, or its child packages, if any. | | | | | | | | | |

This rule can be partially enforced by the GNATcheck switches Visible\_Components applied.

## References

[1] “SEI CERT C Coding Standard,” The Software Engineering Institute.

[2] “Guidelines for the Use of the C Language in Critical Systems,” MISRA, 2013.

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

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

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

[6] “ISO/IEC JTC 1/SC 22/WG9 Ada Reference Manual-Language and Standard

Libraries-ISO/IEC 8652:2020”, publication expected in 2020.

[7] High-Integrity Object-Oriented Programming in Ada, Version 1.4, AdaCore,

October 24, 2016.

[8] SPARK 2014 User’s Guide, AdaCore,

<http://docs.adacore.com/spark2014-docs/html/ug/index.html>

[9] GNAT User’s Guide for Native Platforms, AdaCore,

<http://docs.adacore.com/live/wave/gnat_ugn/html/gnat_ugn/gnat_ugn.html>

[10] B. Liskov and J. Wing. “A behavioral notion of subtyping”,ACM Transactions

on Programming Languages and Systems (TOPLAS), Vol. 16, Issue 6

(November 1994), pp 1811-1841.

[11] RTCA DO-178C/EUROCAE ED-12C.Software Considerations in

Airborne Systems and Equipment Certification. December 2011

[12] B. Meyer. “Object-Oriented Software Construction”, Prentice Hall Professional

Technical Reference, 2nd Edition, 1997.

[13] GNATstack User’s Guide, AdaCore,

<http://docs.adacore.com/live/wave/gnatstack/html/gnatstack_ug/index.html>

[14] Common Weakness Enumeration (CWE), MITRE, 2019

[15] “SEI CERT Oracle Coding Standard for Java,” The Software Engineering Institute.

# 

# AdaCore Documentary License

2018-10-03

These AdaCore Licensing Terms (the “License”) set forth the terms under which AdaCore grants a non-exclusive, worldwide, perpetual license for material delivered by AdaCore to the Customer under a Special Development contract.

## Definition of the Material

Unless agreed otherwise, this license covers any deliverable provided by AdaCore under a Special Development contract that qualifies as documentation, certification or qualification material (collectively “AdaCore Material”).

This license does not apply to software deliverables, which are subject to the “AdaCore Software License”.

## License

AdaCore Material is licensed, under any of the following terms at the option of the Customer, or under alternative licensing conditions that can be found with such Material:

1. the GNU General Public License (GPL) version 3, or any later version published by the FSF, or
2. the GNU Free Documentation License version 1.3, or any later license published by the FSF, or
3. the Creative Commons Attribution, Share-Alike license, version 4.0, as published by the Creative Commons Foundation, or
4. any other terms of the Customer’s choosing, provided that:
   1. AdaCore Material is incorporated in whole or part in a document containing significant content other than such AdaCore Material; and
   2. an acknowledgement is made that parts of the document were produced by AdaCore; and
   3. AdaCore is not identified as author of the whole document; and
   4. Warranty and liability disclaimers contained in AdaCore Material are preserved;
   5. such other terms, if they permit redistribution of this document to third parties, require that any further redistribution satisfies (b), (c), and (d) above.

AdaCore represents and warrants to the Customer that it has the right to distribute AdaCore Material under the terms in this License.

# GNU General Public License

GNU GENERAL PUBLIC LICENSE

Version 3, 29 June 2007

Copyright (C) 2007 Free Software Foundation, Inc. <http://fsf.org/>

Everyone is permitted to copy and distribute verbatim copies

of this license document, but changing it is not allowed.

Preamble

The GNU General Public License is a free, copyleft license for

software and other kinds of works.

The licenses for most software and other practical works are designed

to take away your freedom to share and change the works. By contrast,

the GNU General Public License is intended to guarantee your freedom to

share and change all versions of a program--to make sure it remains free

software for all its users. We, the Free Software Foundation, use the

GNU General Public License for most of our software; it applies also to

any other work released this way by its authors. You can apply it to

your programs, too.

When we speak of free software, we are referring to freedom, not

price. Our General Public Licenses are designed to make sure that you

have the freedom to distribute copies of free software (and charge for

them if you wish), that you receive source code or can get it if you

want it, that you can change the software or use pieces of it in new

free programs, and that you know you can do these things.

To protect your rights, we need to prevent others from denying you

these rights or asking you to surrender the rights. Therefore, you have

certain responsibilities if you distribute copies of the software, or if

you modify it: responsibilities to respect the freedom of others.

For example, if you distribute copies of such a program, whether

gratis or for a fee, you must pass on to the recipients the same

freedoms that you received. You must make sure that they, too, receive

or can get the source code. And you must show them these terms so they

know their rights.

Developers that use the GNU GPL protect your rights with two steps:

(1) assert copyright on the software, and (2) offer you this License

giving you legal permission to copy, distribute and/or modify it.

For the developers' and authors' protection, the GPL clearly explains

that there is no warranty for this free software. For both users' and

authors' sake, the GPL requires that modified versions be marked as

changed, so that their problems will not be attributed erroneously to

authors of previous versions.

Some devices are designed to deny users access to install or run

modified versions of the software inside them, although the manufacturer

can do so. This is fundamentally incompatible with the aim of

protecting users' freedom to change the software. The systematic

pattern of such abuse occurs in the area of products for individuals to

use, which is precisely where it is most unacceptable. Therefore, we

have designed this version of the GPL to prohibit the practice for those

products. If such problems arise substantially in other domains, we

stand ready to extend this provision to those domains in future versions

of the GPL, as needed to protect the freedom of users.

Finally, every program is threatened constantly by software patents.

States should not allow patents to restrict development and use of

software on general-purpose computers, but in those that do, we wish to

avoid the special danger that patents applied to a free program could

make it effectively proprietary. To prevent this, the GPL assures that

patents cannot be used to render the program non-free.

The precise terms and conditions for copying, distribution and

modification follow.

TERMS AND CONDITIONS

0. Definitions.

"This License" refers to version 3 of the GNU General Public License.

"Copyright" also means copyright-like laws that apply to other kinds of

works, such as semiconductor masks.

"The Program" refers to any copyrightable work licensed under this

License. Each licensee is addressed as "you". "Licensees" and

"recipients" may be individuals or organizations.

To "modify" a work means to copy from or adapt all or part of the work

in a fashion requiring copyright permission, other than the making of an

exact copy. The resulting work is called a "modified version" of the

earlier work or a work "based on" the earlier work.

A "covered work" means either the unmodified Program or a work based

on the Program.

To "propagate" a work means to do anything with it that, without

permission, would make you directly or secondarily liable for

infringement under applicable copyright law, except executing it on a

computer or modifying a private copy. Propagation includes copying,

distribution (with or without modification), making available to the

public, and in some countries other activities as well.

To "convey" a work means any kind of propagation that enables other

parties to make or receive copies. Mere interaction with a user through

a computer network, with no transfer of a copy, is not conveying.

An interactive user interface displays "Appropriate Legal Notices"

to the extent that it includes a convenient and prominently visible

feature that (1) displays an appropriate copyright notice, and (2)

tells the user that there is no warranty for the work (except to the

extent that warranties are provided), that licensees may convey the

work under this License, and how to view a copy of this License. If

the interface presents a list of user commands or options, such as a

menu, a prominent item in the list meets this criterion.

1. Source Code.

The "source code" for a work means the preferred form of the work

for making modifications to it. "Object code" means any non-source

form of a work.

A "Standard Interface" means an interface that either is an official

standard defined by a recognized standards body, or, in the case of

interfaces specified for a particular programming language, one that

is widely used among developers working in that language.

The "System Libraries" of an executable work include anything, other

than the work as a whole, that (a) is included in the normal form of

packaging a Major Component, but which is not part of that Major

Component, and (b) serves only to enable use of the work with that

Major Component, or to implement a Standard Interface for which an

implementation is available to the public in source code form. A

"Major Component", in this context, means a major essential component

(kernel, window system, and so on) of the specific operating system

(if any) on which the executable work runs, or a compiler used to

produce the work, or an object code interpreter used to run it.

The "Corresponding Source" for a work in object code form means all

the source code needed to generate, install, and (for an executable

work) run the object code and to modify the work, including scripts to

control those activities. However, it does not include the work's

System Libraries, or general-purpose tools or generally available free

programs which are used unmodified in performing those activities but

which are not part of the work. For example, Corresponding Source

includes interface definition files associated with source files for

the work, and the source code for shared libraries and dynamically

linked subprograms that the work is specifically designed to require,

such as by intimate data communication or control flow between those

subprograms and other parts of the work.

The Corresponding Source need not include anything that users

can regenerate automatically from other parts of the Corresponding

Source.

The Corresponding Source for a work in source code form is that

same work.

2. Basic Permissions.

All rights granted under this License are granted for the term of

copyright on the Program, and are irrevocable provided the stated

conditions are met. This License explicitly affirms your unlimited

permission to run the unmodified Program. The output from running a

covered work is covered by this License only if the output, given its

content, constitutes a covered work. This License acknowledges your

rights of fair use or other equivalent, as provided by copyright law.

You may make, run and propagate covered works that you do not

convey, without conditions so long as your license otherwise remains

in force. You may convey covered works to others for the sole purpose

of having them make modifications exclusively for you, or provide you

with facilities for running those works, provided that you comply with

the terms of this License in conveying all material for which you do

not control copyright. Those thus making or running the covered works

for you must do so exclusively on your behalf, under your direction

and control, on terms that prohibit them from making any copies of

your copyrighted material outside their relationship with you.

Conveying under any other circumstances is permitted solely under

the conditions stated below. Sublicensing is not allowed; section 10

makes it unnecessary.

3. Protecting Users' Legal Rights From Anti-Circumvention Law.

No covered work shall be deemed part of an effective technological

measure under any applicable law fulfilling obligations under article

11 of the WIPO copyright treaty adopted on 20 December 1996, or

similar laws prohibiting or restricting circumvention of such

measures.

When you convey a covered work, you waive any legal power to forbid

circumvention of technological measures to the extent such circumvention

is effected by exercising rights under this License with respect to

the covered work, and you disclaim any intention to limit operation or

modification of the work as a means of enforcing, against the work's

users, your or third parties' legal rights to forbid circumvention of

technological measures.

4. Conveying Verbatim Copies.

You may convey verbatim copies of the Program's source code as you

receive it, in any medium, provided that you conspicuously and

appropriately publish on each copy an appropriate copyright notice;

keep intact all notices stating that this License and any

non-permissive terms added in accord with section 7 apply to the code;

keep intact all notices of the absence of any warranty; and give all

recipients a copy of this License along with the Program.

You may charge any price or no price for each copy that you convey,

and you may offer support or warranty protection for a fee.

5. Conveying Modified Source Versions.

You may convey a work based on the Program, or the modifications to

produce it from the Program, in the form of source code under the

terms of section 4, provided that you also meet all of these conditions:

a) The work must carry prominent notices stating that you modified

it, and giving a relevant date.

b) The work must carry prominent notices stating that it is

released under this License and any conditions added under section

7. This requirement modifies the requirement in section 4 to

"keep intact all notices".

c) You must license the entire work, as a whole, under this

License to anyone who comes into possession of a copy. This

License will therefore apply, along with any applicable section 7

additional terms, to the whole of the work, and all its parts,

regardless of how they are packaged. This License gives no

permission to license the work in any other way, but it does not

invalidate such permission if you have separately received it.

d) If the work has interactive user interfaces, each must display

Appropriate Legal Notices; however, if the Program has interactive

interfaces that do not display Appropriate Legal Notices, your

work need not make them do so.

A compilation of a covered work with other separate and independent

works, which are not by their nature extensions of the covered work,

and which are not combined with it such as to form a larger program,

in or on a volume of a storage or distribution medium, is called an

"aggregate" if the compilation and its resulting copyright are not

used to limit the access or legal rights of the compilation's users

beyond what the individual works permit. Inclusion of a covered work

in an aggregate does not cause this License to apply to the other

parts of the aggregate.

6. Conveying Non-Source Forms.

You may convey a covered work in object code form under the terms

of sections 4 and 5, provided that you also convey the

machine-readable Corresponding Source under the terms of this License,

in one of these ways:

a) Convey the object code in, or embodied in, a physical product

(including a physical distribution medium), accompanied by the

Corresponding Source fixed on a durable physical medium

customarily used for software interchange.

b) Convey the object code in, or embodied in, a physical product

(including a physical distribution medium), accompanied by a

written offer, valid for at least three years and valid for as

long as you offer spare parts or customer support for that product

model, to give anyone who possesses the object code either (1) a

copy of the Corresponding Source for all the software in the

product that is covered by this License, on a durable physical

medium customarily used for software interchange, for a price no

more than your reasonable cost of physically performing this

conveying of source, or (2) access to copy the

Corresponding Source from a network server at no charge.

c) Convey individual copies of the object code with a copy of the

written offer to provide the Corresponding Source. This

alternative is allowed only occasionally and noncommercially, and

only if you received the object code with such an offer, in accord

with subsection 6b.

d) Convey the object code by offering access from a designated

place (gratis or for a charge), and offer equivalent access to the

Corresponding Source in the same way through the same place at no

further charge. You need not require recipients to copy the

Corresponding Source along with the object code. If the place to

copy the object code is a network server, the Corresponding Source

may be on a different server (operated by you or a third party)

that supports equivalent copying facilities, provided you maintain

clear directions next to the object code saying where to find the

Corresponding Source. Regardless of what server hosts the

Corresponding Source, you remain obligated to ensure that it is

available for as long as needed to satisfy these requirements.

e) Convey the object code using peer-to-peer transmission, provided

you inform other peers where the object code and Corresponding

Source of the work are being offered to the general public at no

charge under subsection 6d.

A separable portion of the object code, whose source code is excluded

from the Corresponding Source as a System Library, need not be

included in conveying the object code work.

A "User Product" is either (1) a "consumer product", which means any

tangible personal property which is normally used for personal, family,

or household purposes, or (2) anything designed or sold for incorporation

into a dwelling. In determining whether a product is a consumer product,

doubtful cases shall be resolved in favor of coverage. For a particular

product received by a particular user, "normally used" refers to a

typical or common use of that class of product, regardless of the status

of the particular user or of the way in which the particular user

actually uses, or expects or is expected to use, the product. A product

is a consumer product regardless of whether the product has substantial

commercial, industrial or non-consumer uses, unless such uses represent

the only significant mode of use of the product.

"Installation Information" for a User Product means any methods,

procedures, authorization keys, or other information required to install

and execute modified versions of a covered work in that User Product from

a modified version of its Corresponding Source. The information must

suffice to ensure that the continued functioning of the modified object

code is in no case prevented or interfered with solely because

modification has been made.

If you convey an object code work under this section in, or with, or

specifically for use in, a User Product, and the conveying occurs as

part of a transaction in which the right of possession and use of the

User Product is transferred to the recipient in perpetuity or for a

fixed term (regardless of how the transaction is characterized), the

Corresponding Source conveyed under this section must be accompanied

by the Installation Information. But this requirement does not apply

if neither you nor any third party retains the ability to install

modified object code on the User Product (for example, the work has

been installed in ROM).

The requirement to provide Installation Information does not include a

requirement to continue to provide support service, warranty, or updates

for a work that has been modified or installed by the recipient, or for

the User Product in which it has been modified or installed. Access to a

network may be denied when the modification itself materially and

adversely affects the operation of the network or violates the rules and

protocols for communication across the network.

Corresponding Source conveyed, and Installation Information provided,

in accord with this section must be in a format that is publicly

documented (and with an implementation available to the public in

source code form), and must require no special password or key for

unpacking, reading or copying.

7. Additional Terms.

"Additional permissions" are terms that supplement the terms of this

License by making exceptions from one or more of its conditions.

Additional permissions that are applicable to the entire Program shall

be treated as though they were included in this License, to the extent

that they are valid under applicable law. If additional permissions

apply only to part of the Program, that part may be used separately

under those permissions, but the entire Program remains governed by

this License without regard to the additional permissions.

When you convey a copy of a covered work, you may at your option

remove any additional permissions from that copy, or from any part of

it. (Additional permissions may be written to require their own

removal in certain cases when you modify the work.) You may place

additional permissions on material, added by you to a covered work,

for which you have or can give appropriate copyright permission.

Notwithstanding any other provision of this License, for material you

add to a covered work, you may (if authorized by the copyright holders of

that material) supplement the terms of this License with terms:

a) Disclaiming warranty or limiting liability differently from the

terms of sections 15 and 16 of this License; or

b) Requiring preservation of specified reasonable legal notices or

author attributions in that material or in the Appropriate Legal

Notices displayed by works containing it; or

c) Prohibiting misrepresentation of the origin of that material, or

requiring that modified versions of such material be marked in

reasonable ways as different from the original version; or

d) Limiting the use for publicity purposes of names of licensors or

authors of the material; or

e) Declining to grant rights under trademark law for use of some

trade names, trademarks, or service marks; or

f) Requiring indemnification of licensors and authors of that

material by anyone who conveys the material (or modified versions of

it) with contractual assumptions of liability to the recipient, for

any liability that these contractual assumptions directly impose on

those licensors and authors.

All other non-permissive additional terms are considered "further

restrictions" within the meaning of section 10. If the Program as you

received it, or any part of it, contains a notice stating that it is

governed by this License along with a term that is a further

restriction, you may remove that term. If a license document contains

a further restriction but permits relicensing or conveying under this

License, you may add to a covered work material governed by the terms

of that license document, provided that the further restriction does

not survive such relicensing or conveying.

If you add terms to a covered work in accord with this section, you

must place, in the relevant source files, a statement of the

additional terms that apply to those files, or a notice indicating

where to find the applicable terms.

Additional terms, permissive or non-permissive, may be stated in the

form of a separately written license, or stated as exceptions;

the above requirements apply either way.

8. Termination.

You may not propagate or modify a covered work except as expressly

provided under this License. Any attempt otherwise to propagate or

modify it is void, and will automatically terminate your rights under

this License (including any patent licenses granted under the third

paragraph of section 11).

However, if you cease all violation of this License, then your

license from a particular copyright holder is reinstated (a)

provisionally, unless and until the copyright holder explicitly and

finally terminates your license, and (b) permanently, if the copyright

holder fails to notify you of the violation by some reasonable means

prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is

reinstated permanently if the copyright holder notifies you of the

violation by some reasonable means, this is the first time you have

received notice of violation of this License (for any work) from that

copyright holder, and you cure the violation prior to 30 days after

your receipt of the notice.

Termination of your rights under this section does not terminate the

licenses of parties who have received copies or rights from you under

this License. If your rights have been terminated and not permanently

reinstated, you do not qualify to receive new licenses for the same

material under section 10.

9. Acceptance Not Required for Having Copies.

You are not required to accept this License in order to receive or

run a copy of the Program. Ancillary propagation of a covered work

occurring solely as a consequence of using peer-to-peer transmission

to receive a copy likewise does not require acceptance. However,

nothing other than this License grants you permission to propagate or

modify any covered work. These actions infringe copyright if you do

not accept this License. Therefore, by modifying or propagating a

covered work, you indicate your acceptance of this License to do so.

10. Automatic Licensing of Downstream Recipients.

Each time you convey a covered work, the recipient automatically

receives a license from the original licensors, to run, modify and

propagate that work, subject to this License. You are not responsible

for enforcing compliance by third parties with this License.

An "entity transaction" is a transaction transferring control of an

organization, or substantially all assets of one, or subdividing an

organization, or merging organizations. If propagation of a covered

work results from an entity transaction, each party to that

transaction who receives a copy of the work also receives whatever

licenses to the work the party's predecessor in interest had or could

give under the previous paragraph, plus a right to possession of the

Corresponding Source of the work from the predecessor in interest, if

the predecessor has it or can get it with reasonable efforts.

You may not impose any further restrictions on the exercise of the

rights granted or affirmed under this License. For example, you may

not impose a license fee, royalty, or other charge for exercise of

rights granted under this License, and you may not initiate litigation

(including a cross-claim or counterclaim in a lawsuit) alleging that

any patent claim is infringed by making, using, selling, offering for

sale, or importing the Program or any portion of it.

11. Patents.

A "contributor" is a copyright holder who authorizes use under this

License of the Program or a work on which the Program is based. The

work thus licensed is called the contributor's "contributor version".

A contributor's "essential patent claims" are all patent claims

owned or controlled by the contributor, whether already acquired or

hereafter acquired, that would be infringed by some manner, permitted

by this License, of making, using, or selling its contributor version,

but do not include claims that would be infringed only as a

consequence of further modification of the contributor version. For

purposes of this definition, "control" includes the right to grant

patent sublicenses in a manner consistent with the requirements of

this License.

Each contributor grants you a non-exclusive, worldwide, royalty-free

patent license under the contributor's essential patent claims, to

make, use, sell, offer for sale, import and otherwise run, modify and

propagate the contents of its contributor version.

In the following three paragraphs, a "patent license" is any express

agreement or commitment, however denominated, not to enforce a patent

(such as an express permission to practice a patent or covenant not to

sue for patent infringement). To "grant" such a patent license to a

party means to make such an agreement or commitment not to enforce a

patent against the party.

If you convey a covered work, knowingly relying on a patent license,

and the Corresponding Source of the work is not available for anyone

to copy, free of charge and under the terms of this License, through a

publicly available network server or other readily accessible means,

then you must either (1) cause the Corresponding Source to be so

available, or (2) arrange to deprive yourself of the benefit of the

patent license for this particular work, or (3) arrange, in a manner

consistent with the requirements of this License, to extend the patent

license to downstream recipients. "Knowingly relying" means you have

actual knowledge that, but for the patent license, your conveying the

covered work in a country, or your recipient's use of the covered work

in a country, would infringe one or more identifiable patents in that

country that you have reason to believe are valid.

If, pursuant to or in connection with a single transaction or

arrangement, you convey, or propagate by procuring conveyance of, a

covered work, and grant a patent license to some of the parties

receiving the covered work authorizing them to use, propagate, modify

or convey a specific copy of the covered work, then the patent license

you grant is automatically extended to all recipients of the covered

work and works based on it.

A patent license is "discriminatory" if it does not include within

the scope of its coverage, prohibits the exercise of, or is

conditioned on the non-exercise of one or more of the rights that are

specifically granted under this License. You may not convey a covered

work if you are a party to an arrangement with a third party that is

in the business of distributing software, under which you make payment

to the third party based on the extent of your activity of conveying

the work, and under which the third party grants, to any of the

parties who would receive the covered work from you, a discriminatory

patent license (a) in connection with copies of the covered work

conveyed by you (or copies made from those copies), or (b) primarily

for and in connection with specific products or compilations that

contain the covered work, unless you entered into that arrangement,

or that patent license was granted, prior to 28 March 2007.

Nothing in this License shall be construed as excluding or limiting

any implied license or other defenses to infringement that may

otherwise be available to you under applicable patent law.

12. No Surrender of Others' Freedom.

If conditions are imposed on you (whether by court order, agreement or

otherwise) that contradict the conditions of this License, they do not

excuse you from the conditions of this License. If you cannot convey a

covered work so as to satisfy simultaneously your obligations under this

License and any other pertinent obligations, then as a consequence you may

not convey it at all. For example, if you agree to terms that obligate you

to collect a royalty for further conveying from those to whom you convey

the Program, the only way you could satisfy both those terms and this

License would be to refrain entirely from conveying the Program.

13. Use with the GNU Affero General Public License.

Notwithstanding any other provision of this License, you have

permission to link or combine any covered work with a work licensed

under version 3 of the GNU Affero General Public License into a single

combined work, and to convey the resulting work. The terms of this

License will continue to apply to the part which is the covered work,

but the special requirements of the GNU Affero General Public License,

section 13, concerning interaction through a network will apply to the

combination as such.

14. Revised Versions of this License.

The Free Software Foundation may publish revised and/or new versions of

the GNU General Public License from time to time. Such new versions will

be similar in spirit to the present version, but may differ in detail to

address new problems or concerns.

Each version is given a distinguishing version number. If the

Program specifies that a certain numbered version of the GNU General

Public License "or any later version" applies to it, you have the

option of following the terms and conditions either of that numbered

version or of any later version published by the Free Software

Foundation. If the Program does not specify a version number of the

GNU General Public License, you may choose any version ever published

by the Free Software Foundation.

If the Program specifies that a proxy can decide which future

versions of the GNU General Public License can be used, that proxy's

public statement of acceptance of a version permanently authorizes you

to choose that version for the Program.

Later license versions may give you additional or different

permissions. However, no additional obligations are imposed on any

author or copyright holder as a result of your choosing to follow a

later version.

15. Disclaimer of Warranty.

THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY

APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT

HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY

OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO,

THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR

PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM

IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF

ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

16. Limitation of Liability.

IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING

WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MODIFIES AND/OR CONVEYS

THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY

GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE

USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF

DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD

PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS),

EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF

SUCH DAMAGES.

17. Interpretation of Sections 15 and 16.

If the disclaimer of warranty and limitation of liability provided

above cannot be given local legal effect according to their terms,

reviewing courts shall apply local law that most closely approximates

an absolute waiver of all civil liability in connection with the

Program, unless a warranty or assumption of liability accompanies a

copy of the Program in return for a fee.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest

possible use to the public, the best way to achieve this is to make it

free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest

to attach them to the start of each source file to most effectively

state the exclusion of warranty; and each file should have at least

the "copyright" line and a pointer to where the full notice is found.

<one line to give the program's name and a brief idea of what it does.>

Copyright (C) <year> <name of author>

This program is free software: you can redistribute it and/or modify

it under the terms of the GNU General Public License as published by

the Free Software Foundation, either version 3 of the License, or

(at your option) any later version.

This program is distributed in the hope that it will be useful,

but WITHOUT ANY WARRANTY; without even the implied warranty of

MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the

GNU General Public License for more details.

You should have received a copy of the GNU General Public License

along with this program. If not, see <http://www.gnu.org/licenses/>.

Also add information on how to contact you by electronic and paper mail.

If the program does terminal interaction, make it output a short

notice like this when it starts in an interactive mode:

<program> Copyright (C) <year> <name of author>

This program comes with ABSOLUTELY NO WARRANTY; for details type `show w'.

This is free software, and you are welcome to redistribute it

under certain conditions; type `show c' for details.

The hypothetical commands `show w' and `show c' should show the appropriate

parts of the General Public License. Of course, your program's commands

might be different; for a GUI interface, you would use an "about box".

You should also get your employer (if you work as a programmer) or school,

if any, to sign a "copyright disclaimer" for the program, if necessary.

For more information on this, and how to apply and follow the GNU GPL, see

<http://www.gnu.org/licenses/>.

The GNU General Public License does not permit incorporating your program

into proprietary programs. If your program is a subroutine library, you

may consider it more useful to permit linking proprietary applications with

the library. If this is what you want to do, use the GNU Lesser General

Public License instead of this License. But first, please read

<http://www.gnu.org/philosophy/why-not-lgpl.html>.