
High-Integrity Object-Oriented Programming in Ada

Release 1.3

AdaCore

July 25, 2011

CONTENTS

1	Introduction	1
2	Object Orientation Concepts	3
2.1	Fundamentals of Object-Oriented Development	3
2.2	Object-Oriented Programming (OOP)	4
2.3	Additional Object-Oriented Programming Concepts	5
3	Object Orientation in Ada	7
3.1	Classes, Objects, Inheritance, and Polymorphism	7
3.2	Encapsulation Features	10
3.3	Constructors	11
3.4	Finalization and Controlled Types	12
3.5	Multiple Inheritance and Interface Types	12
3.6	Other Object-Oriented features	14
3.7	Support for Programming by Contract in Ada 2012	15
4	Vulnerabilities and Their Mitigation	19
4.1	Lack of Encapsulation	19
4.2	Dynamic Dispatch and Substitutability	20
4.3	Improper Overriding	22
4.4	Multiple Inheritance	24
4.5	Static Binding and Overriding	25
4.6	Memory Management Issues	26
5	Complexity Management	29
5.1	Control Inheritance Depth / Width and Multiple Inheritance	29
5.2	Control Class Coupling	30
5.3	Minimize Need for Ada Run-Time Support and Facilitate Source-to-Object Traceability	32
6	Safety and Verification Considerations	35
6.1	Use of Pre/Post/Invariant Aspects for Verification	35
6.2	Robustness of Dynamic Dispatch Mechanism	38
7	Directions for GNAT Pro Users	39
7.1	Writing a Coding Standard with GNATcheck	39
7.2	Using SPARK Pro for Formal Proofs	41
7.3	Using GNATstack for Stack Resource Analysis	42
7.4	OO Support in GNAT Pro High-Integrity Profiles	44
8	Conclusion	47

INTRODUCTION

This paper's goal is to provide guidance on how to use Ada's Object Oriented (OO) features for High-Integrity (HI) applications; i.e. high-reliability systems with requirements for safety and/or security which may need to demonstrate compliance with domain-specific certification standards. In the past, using OO was not considered an option when developing HI applications which remained as small as possible. Nowadays, such applications have grown in size and can even be very large: several millions of line of code are becoming more common in our community. It becomes therefore more attractive to use OO techniques that are renowned to help maintain a high level of modularity, extensibility, reusability and maintainability in large applications. The paper was written by AdaCore experts with extensive experience in this area through their participation in industrial working groups such as the joint EUROCAE WG71 / RTCA SC 205 working group defining the upcoming RTCA DO178-C/EUROCAE ED12-C avionics safety standard, and ISO's Ada Rapporteur Group that manages the Ada language standardization process. Another source of experience comes from the support of AdaCore customers, in domains such as aerospace and transportation, since the company's inception in the mid 1990s. Many customers are using AdaCore tools during their certification process, and some have already completed the highest level of certification with Ada code while extensively using Object Oriented features.

The *Object Orientation Concepts* chapter summarizes the principal concepts in order to establish the terminology and provide criteria for comparing languages' approaches. The *Object Orientation in Ada* chapter introduces Ada's model for Object Orientation. Readers familiar with Ada 95 can move quickly to the last sections of this chapter, which describe several new features that are being added to the next revision of language, known as Ada 2012, and which are extremely relevant to safe Object-Oriented programming. The next three chapters – *Vulnerabilities and Their Mitigation*, *Complexity Management*, and *Safety and Verification Considerations* – describe known concerns related to using Object Oriented technology in a High Integrity environment and discuss recommended solutions. The final chapter, *Directions for GNAT Pro Users*, starts with a user checklist summarizing the steps we suggest taking when starting out in this area, and then presents the relevant elements and tools of the GNAT Pro technology.

OBJECT ORIENTATION CONCEPTS

Object Orientation is a software design and implementation approach that can contribute towards a robust and maintainable system architecture, especially in applications that are “data rich” (i.e., that need to deal with multiple kinds of data that are related in various ways). The basic elements of Object Orientation first appeared in the Simula 67 language in the late 1960s, gained traction (especially in the research and academic communities) with Smalltalk in the 1970s and 1980s, and found their way into mainstream practice with the emergence of languages such as C++, Ada 95, Java, and C#.

Object Orientation is characterized by a small number of basic concepts, but different languages have their own vocabularies for these concepts. With the goal of remaining language independent, this summary uses terminology that has entered common usage and that appears in UML or in one or more widely-used Object-Oriented languages.

2.1 Fundamentals of Object-Oriented Development

The core of Object Orientation is embodied in three concepts: *class*, *object*, and *encapsulation*.

2.1.1 Class

A *class* is a data type and thus a template for objects; it specifies the *fields* that are present in, and the *operations* that may be invoked on, any object of the class. Such fields and operations are referred to as *instance fields/operations*. A class may also specify so-called *static* fields and/or operations: such fields and operations exist independently of any objects of the class.

A class is also a module, i.e. a unit of program composition. From this perspective the fields and operations are said to be *members* of the class.

Languages differ with respect to whether the class concept is realized by a single feature that serves as both a type and a module, or by separate features. C++ and Java are in the first category; Ada, CLOS, and Smalltalk are in the second. Languages also differ with respect to whether a class’s logical interface is combined with its implementation in the same module or are separated into distinct modules. Java is in the first category; C++ and Ada are in the second.

2.1.2 Object

An *object* is an instance of a class. The set of instance fields for an object is sometimes referred as the *state* of the object. An object thus has *state* (fields) and *behavior* (operations).

2.1.3 Encapsulation

Encapsulation is the programmer’s ability to restrict access to class’s members, for example to ensure that an instance or a static field is visible only within the implementation of the class’s operations, and to provide a

procedural interface to operations for accessing the data. It is generally good practice, in order to prevent certain kinds of coding errors and to reduce the effort in system maintenance as requirements evolve, to minimize the accessibility of state / fields so that they may only be referenced in those parts of the program with a “need to know”.

Several levels of encapsulation are found in practice. The most restrictive is to limit a class member so that it is visible only within the implementation of the class itself. At the other extreme, the most general is to allow a class member to be visible wherever the class itself can be referenced. Intermediate levels are also possible; for example, limiting a class member so that it is visible only within its containing class and all its subclasses.

2.2 Object-Oriented Programming (OOP)

The concepts described above, sometimes collectively referred to as *Object-Oriented Design* or *Object-Based Programming*, are basic to Object-Oriented development and provide a sound basis for effectively specifying a system’s high-level architecture. Ada 83 is an example of a language with features to realize these concepts. However, the power and flexibility of Object Orientation is significantly enhanced through several additional mechanisms: *inheritance*, *polymorphism*, and *dynamic binding*.

2.2.1 Inheritance

Inheritance may be viewed from two angles. From a programming perspective, inheritance is a language feature that allows a programmer to define a new class (the *subclass*) in terms of an existing class (the *superclass*) by adding new state and/or operations, and by overriding existing instance operations with new implementations. Superclass operations that are not overridden, and all fields, are implicitly inherited by (i.e., are members of) the subclass. Inheriting from a superclass is sometimes referred to as *implementation inheritance* or *inheritance of implementation* (since it brings in the superclass’s state and operation code) and is also sometimes called *programming by extension*.

From a type theory (or data modeling) perspective, inheritance is a specialization relationship between a *subtype* (subclass) and a *supertype* (superclass): an instance of the subtype is also an instance of the supertype, and thus a subtype instance should be usable whenever a supertype instance is required. This latter property is sometimes known as the *Liskov Substitution Principle* (LSP) [LW94]. Note that this usage of the term *subtype* is different from the Ada sense.

An *inheritance hierarchy* with respect to a particular class consists of that class together with all of its direct and indirect subclasses.

Inheritance is an effective technique for reuse and system extensibility, but adhering to LSP imposes some constraints on the implementation of a subclass overriding operations. If the superclass operation imposes some precondition, then the subclass’s implementation should not impose a more restrictive precondition. The reason for this rule is that the operation’s *contract* that is known to the caller is the precondition and postcondition for the superclass’s version. If the overriding operation required a more restrictive precondition, then an invocation of the operation that satisfied the superclass’s precondition might violate the subclass’s precondition, resulting in a run-time error. For similar reasons the subclass’s implementation of an overriding operation should not loosen the postcondition that applies to the superclass’s version of the operation.

Languages differ with respect to whether a class may only have one direct superclass (so-called *single inheritance*) or an arbitrary number of direct superclasses (*multiple inheritance*). Ada and Java are examples of languages with single inheritance of implementation; C++ is the principal example of a language with multiple inheritance of implementation. Multiple implementation inheritance brings expressive power but also semantic and implementation complexity. Well-known problems include:

- *Name clashes*. Different superclasses may have fields with the same name, or operations with the same signature. Exactly what is inherited by the subclass, and how is it referenced?
- *“Diamond inheritance”*. A “root” class C0 may define some fields and operations that then get inherited by subclasses C1 and C2. If a class C3 inherits from C1 and C2, exactly what does it inherit indirectly from C0, and how is it referenced?

For further discussion see the *Multiple Inheritance* section in *Vulnerabilities and Their Mitigation* below.

2.2.2 Polymorphism

Polymorphism is the ability of a variable to denote objects from different classes (in the same inheritance hierarchy) at different times. Polymorphism is generally realized through references, since the compiler cannot predict in advance the maximum size for an object that will be denoted by a polymorphic variable. Languages differ with respect to whether the references are explicit (Ada, C++) or implicit (Java), and whether the resultant storage management issues are the responsibility of the programmer (Ada, C++) or an implementation-provided garbage collector (Java).

2.2.3 Dynamic Binding

Dynamic binding is the run-time determination of an operation performed on a polymorphic variable, based on the class of the object that is currently denoted by the variable. This is in contrast to *static binding*, which resolves an operation invocation based on the compile-time type of the variable on which the operation is invoked. Languages differ with respect to how they determine whether an operation is to be bound statically versus dynamically.

The judicious use of inheritance, polymorphism, and dynamic binding can ease system maintenance / extensibility. If a new class is added to an inheritance hierarchy, code that processes objects from existing classes does not have to be modified or even recompiled. In contrast, with a design based on functional decomposition rather than OOP, adding new data generally entails modifying a centralized data structure (variant record) and all the modules that process this data structure.

2.3 Additional Object-Oriented Programming Concepts

The essence of Object-Oriented Programming is captured in the constructs so far described, but several additional concepts are often realized in Object-Oriented Languages.

2.3.1 Abstract Class

An *abstract* class is a class that does not permit instantiation (construction of objects); it serves as the root of an inheritance hierarchy and typically includes operations that are likewise abstract (lacking an implementation). A *concrete* class is one that is not abstract; a concrete class that inherits from an abstract class has to override all of its superclass's abstract operations with concrete versions.

An abstract class is useful for modeling a data space that is to be partitioned. A “root” class is defined as abstract, and concrete subclasses then correspond to the various partitions.

2.3.2 Interface

An *interface* is like a lightweight abstract class, containing only abstract operations and having no instance fields. A class can inherit from (or *implement*) an arbitrary number of interfaces; this is known as *interface inheritance*. Since interfaces lack concrete operations and per-instance state, multiple interface inheritance avoids the semantic complexity of multiple implementation inheritance.

Interfaces support polymorphism and dynamic binding. A polymorphic variable of an interface type can denote an object from any class that directly or indirectly implements the interface. Applying an interface type operation to such a polymorphic variable invokes the version of the operation supplied for the class of the object that the variable denotes.

2.3.3 Constructor

A *constructor* is an operation that is applied to an object immediately after it has been created. Constructors may be parameterized and typically perform initializations that bring the object to a consistent state in preparation for further processing.

2.3.4 Finalization

Finalization is an operation that is applied to an object after the program has no further need to access the object. Finalization generally is responsible for disposing of resources that were used by the object. Whether finalization is responsible for reclaiming the space occupied by dynamically allocated objects depends on the programming language.

OBJECT ORIENTATION IN ADA

This section explains how Object Orientation is implemented in Ada, taking into account the enhancements being adopted in the Ada 2012 language revision.

3.1 Classes, Objects, Inheritance, and Polymorphism

Object-oriented features were added to Ada as part of the first major revision to the Ada standard, completed in 1995 (ISO/IEC 8652/1995). These features permit the creation of classes of types related by an inheritance mechanism called *type derivation*. Polymorphism is supported by the notions of *class-wide types* and *primitive dispatching operations*. The following sections discuss these features, giving simple examples of their use. Later sections give an overview of related Ada features that enhance and build upon Ada's basic object-oriented capabilities.

3.1.1 Type Classes

As mentioned earlier in the *Class* subsection of *Fundamentals of Object-Oriented Development*, Ada follows a class model based on separate features (types declared within package units) rather than a single module-based class construct. Corresponding to the notion of a type hierarchy, Ada has the concept of a *type class*, which consists of a set of types, created by declaring a root type R within a module-like construct called a *package*. Additional types, called *type extensions*, can be declared in the same or separate packages, extending the properties of the root type. Associated with each type is a unique *tag* that distinguishes it from all other types in the class.

Each type in a class is referred to as a *tagged type*, defined by including the keyword **tagged** in the root type's declaration. Here is an example of a root-level tagged type `Shape` declared within a package `Shapes`, with a single primitive operation:

```
package Shapes is
    type Shape is tagged record
        Color : Color_Type;
    end record;

    procedure Set_Color (Obj : in out Shape; Color : Color_Type);
end Shapes;
```

Each class of types has an associated *class-wide type*, defined implicitly along with the class's root type R , and denoted by the notation R' `Class`.

The root type and its type extensions are referred to as *specific types*. Each specific type in the class can have an associated set of data components and operations. In the above example, the tagged type `Shape` is a record type with a single component, `Color`, and one primitive operation, a procedure `Set_Color` with one parameter of the type. The implementation of a type's primitive operations are typically given in the package's separately compiled body.

3.1.2 Inheritance and Type Extensions

A new type can be added to a class by inheriting from a parent tagged type by means of a declaration called a *type extension*. The type declared by a type extension inherits the components and primitive operations of the parent type. The new type is said to *derive* from its parent type. Here's an example of a type that extends from the earlier Shape type:

```
package Polygons is

  subtype Side_Count is Integer range 3 .. Integer'Last;
  type Inches is range 0 .. 100;

  type Polygon is new Shapes.Shape with record
    Number_Of_Sides : Side_Count;
    Length_Of_Side   : Inches;
  end record;

  procedure Set_Number_Of_Sides (P : in out Polygon; Sides : Side_Count);
  procedure Set_Length_Of_Side  (P : in out Polygon; Length : Inches);
  function Area (P : Polygon) return Float;

end Polygons;
```

The declaration of type Polygon indicates that it derives from the parent Shape type, thus inheriting the Color component and the primitive operation Set_Color. The inherited components and operations are said to be implicitly declared for the type extension.

A new component Number_Of_Sides is declared for the type, along with a new operation Set_Number_Of_Sides. If Polygon is later extended by another type, that type will further inherit all of Polygon's components and operations.

An instance of a tagged type is created by declaring an object of the type within a package, or within the body of a subprogram (just as for other types in Ada). The object can then be referenced by its name. Tagged objects can also be created dynamically by executing an *allocator*; for example,

```
Ref := new Polygon;
```

which produces a reference to the object (called an *access value* in Ada).

Given an object of type Polygon, its components and operations can be referenced as follows:

```
use Polygons;  -- Open visibility to declarations within package Polygons

Rectangle : Polygon :=
  (Color => Red, Number_Of_Sides => 4, Length_Of_Side => 1.5);
  -- Constructs a value of type Polygon

C : Color_Type := Rectangle.Color;  -- Select the Color component
A : Float := Area (Rectangle);

Set_Color (Rectangle, Color => Green);

Rectangle.Set_Color (Green);  -- Equivalent to the preceding call
Rectangle.Set_Number_Of_Sides (Rectangle.Number_Of_Sides + 1);
```

Note that primitive operations can be invoked either using conventional call notation, passing the tagged object as a parameter, or by using a prefixed form of call, where the object is prefixed to the operation name (this form requires that the tagged parameter is defined as the first formal parameter of the operation).

In the case of operations, an inherited operation can be *overridden* for the type extension. For example:

```
type Square is new Polygon with record ... end record;

overriding
function Area (R : Square) return Float;  -- overrides inherited Area
```

An optional *overriding indicator* can be included with the overriding declaration, which helps to catch errors in cases where the declaration is mistyped. Compilers (such as GNAT) will warn if the overriding indicator is missing. For further discussion see the sections *Improper Overriding* and *Writing a Coding Standard with GNATcheck* later in this document.

3.1.3 Polymorphism and Dynamic Binding

As discussed in the previous section, an instance of a specific tagged type is created by declaring an object of the type. It is also possible to declare an object of the associated class-wide type, though such an object must be initialized from an object of a specific type in the class of types associated with the class-wide type. For example:

```
Some_Shape : Shape'Class := Rectangle;
```

However, once a class-wide object has been created, its tag is fixed, so it's not possible to reassign from an object of another specific tagged type. Thus, such an object is not polymorphic in the sense of being able to take on the value of any object in the type class. Usually it's necessary to introduce a reference type (called an *access type* in Ada) whose values can designate any object in the type class. Access variables can be freely reassigned and embedded within objects, permitting the creation of heterogeneous linked structures in the traditional way. For example:

```
type Any_Shape is access Shape'Class;
Any : Any_Shape := new Square;
```

```
Any := Some_Other_Shape;
```

Polymorphic operations can be declared that take any object in a type class as a parameter:

```
procedure Print_Shape (S : Shape'Class);
```

or that take an access to any object as a parameter:

```
procedure Print_Shape (S : access Shape'Class);
```

The implementation of a class-wide operation will normally be general enough to apply to any object within the type class, and so will typically include calls to common operations of the class that involve dynamic binding.

In Ada, dynamic binding is supported by calling primitive operations of the the root type of a type class and passing an object of the class-wide type as a so-called *controlling operand*. Such a call is referred to as a *dispatching call*. Rather than invoking the root type's operation, a dispatching call will generally result in a call to the operation of the type corresponding to the tag associated with the controlling operand. As an example:

```
procedure Describe_Polygon (P : Polygon'Class) is
    Area : Float := P.Area;  -- Dispatching function call
begin
    Put ("The polygon has" & Integer'Image (P.Number_Of_Sides) & " sides");
    Put_Line (" and an area of" & Float'Image (Area) & " square inches");
end Describe_Polygon;
```

Since each Polygon has a Number_Of_Sides field, that field can be selected from an object of type Polygon'Class (no dispatching is involved). The call to Area dispatches to the particular Area operation associated with the tag of the underlying object (or to the root type's Area operation, if it has not been overridden). Note that, in this example, the prefixed form of call is used for the dispatching call to Area.

A tagged type can be declared as an *abstract type*, in which case its operations are usually declared as abstract. This can make sense when the type serves as a model for an abstraction that has no default implementation. Each concrete type that extends the abstract type must provide the implementation by overriding each of the abstract operations inherited from the abstract parent type. No instances can be created for an abstract type, only for descendant types, and calls to abstract operations are necessarily always dispatching calls. This ensures that there is no “dangling dispatching”; that is, every dispatching call will dispatch to some concrete operation body.

As an example, the type Shape could be declared as abstract, with an abstract Area operation:

```
package Abstract_Shapes is

    type Shape is abstract tagged null record; -- No fields
    function Area (S : Shape) return Float is abstract; -- Always dispatching

end Abstract_Shapes;
```

Abstract types are also allowed to have operations that are null procedures, if it makes sense for the operation to do nothing as a default behavior. Null procedures do not need to be overridden.

Several predefined operations are available for tagged types. Membership tests can be used to determine whether a class-wide object belongs to a subclass of a given class:

```
if Some_Shape in Square'Class then ...
```

Tagged objects can also be assigned and compared for equality, as long as the type is not a limited type (see *Encapsulation Features* below). If a class-wide object is assigned, then the tag of the source object must match the tag of the target, otherwise an exception is raised. Class-wide objects will only compare as equal if they have the same tag and they are component-wise equal.

3.2 Encapsulation Features

Ada provides several features that support encapsulation of type properties.

At the module level, packages are partitioned into two main parts, called the *package specification* and *package body*, which can be physically separate when declared at the top level of the program. Packages can also be nested within packages and other forms of program units. The body of a package contains the implementation of subprograms declared within the package's specification, and can also include the declaration of additional types and data structures needed only by the implementation.

The package specification consists of a *visible part* and an optional *private part*. Entities declared within the visible part are available to all clients of the package (i.e., units that *with* the package), whereas entities declared in the private part are visible only within the visibility region encompassed by the package itself (which can include parts of its so-called *child units*, as explained below). Thus, a package is a unit that supports division of an abstraction into a well-defined specification of externally visible properties clearly separated from its implementation.

Types themselves can be defined in terms of a visible part and private part, allowing the hiding of type properties that are strictly part of the implementation of an abstraction. A private type is declared in the visible part of its package's specification, and its corresponding full type is declared in the package's private part. The full type might be tagged while the private view is tagged or untagged, depending on whether the abstraction designer intends the type to be extended by clients.

Here is an example based on the earlier *Shape* type, where the type is defined as a tagged private type:

```
package Shapes is

    type Shape is tagged private;
    procedure Set_Color (Obj : in out Shape; Color : Color_Type);

private

    type Shape is tagged record
        Color : Color_Type;
    end record;

end Shapes;
```

A package can be partitioned into multiple units, generally packages, as a kind of subsystem hierarchy, by the use of *child units*. A child unit is either a *public child* or a *private child*. The private part (if any) and body of each public child unit *P.C* of a package *P* has access to the private part of its parent package *P*. This permits an abstraction to be decomposed into separate interfaces and implementations, which can help to organize the

abstraction and better manage its complexity. The full specification and body of a private child has access to its parent's private part, effectively making the private child a part of the package hierarchy's implementation. A private child can only be `withed` from units that likewise have visibility into the private part of the parent (i.e., only within the unit hierarchy rooted at the parent).

We can add a child unit of package `Shapes` as follows:

```
package Shapes.Polygons is

  type Polygon is new Shape with private;

  procedure Set_Number_Of_Sides (P : in out Polygon; Sides : Side_Count);
  procedure Set_Length_Of_Side  (P : in out Polygon; Length : Inches);
  function Area (P : Polygon) return Float;

private

  type Polygon is new Shape with record
    Number_Of_Sides : Side_Count;
    Length_Of_Side  : Inches;
  end record;

end Shapes.Polygons;
```

Note that the type `Polygon` is declared as deriving from type `Shape` with a private extension. This allows the details of the type's components to be hidden from clients of the package, but clients are still allowed to extend the type and be aware that such extensions are part of the `Shape'Class` type hierarchy. Note that the implementation of the `Polygons` child package has access to the details defined in the private part of its parent package.

In addition to declaring a type as private, types can be declared to be *limited*, which further restricts the operations that can be performed on objects by clients of the type. Specifically, a limited type does not have the operations of assignment and equality available (although the programmer can explicitly declare an equality operation if this is appropriate for the type). This level of control is commonly needed for abstractions where copying does not make sense, or where copying would compromise safety of data, such as for file descriptors or locked data structures.

3.3 Constructors

Object-oriented languages generally provide some means of initializing an object to a well-defined state when its created, via a mechanism commonly called a *constructor*. Ada has several ways of initializing objects. One is by using default initializing of components, which is accomplished by specifying specific default values for one or more components as part of a record type declaration, for example:

```
type Polygon is new Shape with record
  Number_Of_Sides : Side_Count := Side_Count'First;
  Length_Of_Side  : Inches := 0.0;
end record;
```

When an object of this type is declared without explicit initialization, the components object will be initialized with the specified values. This approach is appropriate when there is a default that makes sense for all objects, but it is not too flexible. A more general form of default initialization can be achieved using *controlled types*, which are described in *Finalization and Controlled Types* below.

Another approach is to initialize an object with an *aggregate*, which is a construct that allows a value to be given to each component of the object as part of the object's initialization:

```
Pentagon : Polygon :=
  (Color => White, Number_Of_Side => 5, Length_Of_Sides => 1.0);
```

This works when all of a type's components are visible, but cannot be used if there are any components not visible at the point where the object is declared (such as when the type is private or is a private extension).

The most general approach to constructing an object's value is to define an appropriately parameterized function that returns a value of the given type. This is usually the best technique when defining a private abstraction, where the details of the implementation are hidden:

```
Pentagon : Polygon := Init_Polygon (White, 5, 1.0);
```

Note that if such a constructor function is declared as a primitive operation, it cannot be inherited and used by type extensions, but must be overridden by each extension. This rule helps ensure proper initialization of fields added in the type extension. It is often undesirable for a constructor to be a primitive operation since there is never any need to dispatch on construction of an object. One can avoid constructor functions to be primitive by declaring them in a subpackage:

```
package Shapes.Polygons is

  type Polygon is new Shape with private;

  package Constructor is
    function Init_Polygon (
      Color           : Color_Type;
      Number_Of_Sides : Side_Count;
      Length_Of_Side  : Inches)
      return Polygon;
  end Constructor;
  ...
```

It is also possible to combine aspects of the different constructor features. For example, for a type that derives from a private extension, and adds some components, an object of the type can be initialized by a special form of aggregate called an *extension aggregate*, that uses default initialization or calls a function to initialize the hidden components and then explicitly initializes any components added by the extension:

```
type Named_Polygon is new Polygon with record
  Name : String (1 .. 15);
end record;

NP : Named_Polygon :=
  (Init_Polygon (Red, 4, 1.0) with Name => "I am a square ");
```

3.4 Finalization and Controlled Types

Most object-oriented languages provide a way to clean up resources associated with an object after the object is no longer needed, in a process called finalization. In Ada, this capability is achieved using the *controlled type* feature, which builds upon on tagged types. The predefined package `Ada.Finalization` defines types `Controlled` and `Limited_Controlled` along with `Initialize` and `Finalize` procedures for each type. A type that extends one of the predefined controlled types can override the inherited `Initialize` and `Finalize` procedures, implementing appropriate default initialization and finalization actions. `Initialize` is implicitly called for objects of such a type when an object is created without an explicit initial value. `Finalize` is implicitly called at the end of an object's lifetime.

Note that deallocation of the space occupied by a controlled object is not tied to its finalization. An object's space is freed up either when it goes out of scope (such as when exiting a subprogram) or when it's explicitly deallocated by the program (Ada is not a garbage-collected language). For types that extend the non-limited type `Controlled`, an additional inherited operation named `Adjust` can be used to support copying of objects that have a *deep* state that needs to be copied, thus providing a user-defined assignment capability.

3.5 Multiple Inheritance and Interface Types

Ada supports multiple inheritance of interfaces; the more general form, multiple inheritance of implementation, is not allowed.

Ada defines a special form of tagged type called an *interface type*, or more simply an *interface*. An interface is essentially an abstract type with no components and with each operation either abstract or null. A type extension is allowed to inherit from one or more interface types (as well as from at most one parent type that is not an interface type).

Here is an example showing multiple inheritance from one regular tagged type and two interface types:

```
package Encryptable_Pkg is
  type Encryptable is interface;

  procedure Encrypt (Item : in out Encryptable) is abstract;
  procedure Decrypt (Item : in out Encryptable) is abstract;
end Encryptable_Pkg;

package Textable_Pkg is
  type Textable is interface;

  function Image (T : Textable) return String is abstract;
  function Value (S : in String) return Textable is abstract;
end Textable_Pkg;

package Person_Pkg is
  type Person is tagged private;
  ... -- Subprograms for Person
private
  ...
end Person_Pkg;

with Encryptable_Pkg, Textable_Pkg;
use Encryptable_Pkg, Textable_Pkg;
package Person_Pkg_Secret_Agent_Pkg is
  type Secret_Agent is new Person and Encryptable and Textable with private;

  ... -- Overriding versions of Person subprograms as needed
  ... -- New subprograms for Secret_Agent

  procedure Encrypt (Item : in out Secret_Agent);
  procedure Decrypt (Item : in out Secret_Agent);
  function Image (T : Secret_Agent) return String;
  function Value (S : in String) return Secret_Agent;
private
  ...
end Person_Pkg_Secret_Agent_Pkg;
```

Here is another example. Consider a queue abstraction:

```
type Queue is limited interface;

procedure Enqueue (Q : in out Queue; Item : Item_Type) is abstract;
procedure Dequeue (Q : in out Queue; Item : Item_Type) is abstract;

-- ... other abstract operations such as Count
```

A type that implements the `Queue` interface must override all of `Queue` abstract operations.

An interface can be declared as *synchronized*, which means that any type that implements it must be a concurrent type (that is, a task or protected type). As an example, consider a synchronized `Queue` interface type. This interface could add additional forms of dequeuing operation that would wait until an item is available rather than raising an exception if the queue is empty:

```
type Synch_Queue is synchronized interface and Queue;
```

```
procedure Enqueue (Q : in out Queue; Item : Item_Type) is abstract;
procedure Dequeue (Q : in out Queue; Item : Item_Type) is abstract;

procedure Dequeue_When_Ready
  (Q : in out Queue; Item : Item_Type) is abstract;
```

This type can be implemented by a task type (or protected type), ensuring safe concurrent access to the queue:

```
task type Concurrent_Queue is new Synch_Queue with

  entry Enqueue (Item : Item_Type);
  entry Dequeue (Item : Item_Type);
  entry Dequeue_When_Ready (Item : Item_Type);

end Concurrent_Queue;
```

Note that the inherited queue operations are overridden by entries that only take one parameter (because the queue object is a task that will be given as the prefix in an entry call, and is implicit for the entries themselves).

```
package Encryptable_Pkg is
  type Encryptable is interface;

  procedure Encrypt (Item : in out Encryptable) is abstract;
  procedure Decrypt (Item : in out Encryptable) is abstract;
end Encryptable_Pkg;
```

```
package Textable_Pkg is
  type Textable is interface;

  function Image (T : Textable) return String is abstract;
  function Value (S : in String) return Textable is abstract;
end Textable_Pkg;
```

```
package Person_Pkg is
  type Person is tagged private;
  ... -- Subprograms for Person
private
  ...
end Person_Pkg;

with Encryptable_Pkg, Textable_Pkg;
use Encryptable_Pkg, Textable_Pkg;
package Person_Pkg_Secret_Agent_Pkg is
  type Secret_Agent is new Person and Encryptable and Textable with private;

  ... -- Overriding versions of Person subprograms as needed
  ... -- New subprograms for Secret_Agent

  procedure Encrypt (Item : in out Secret_Agent);
  procedure Decrypt (Item : in out Secret_Agent);
  function Image (T : Secret_Agent) return String;
  function Value (S : in String) return Secret_Agent;
private
  ...
end Person_Pkg_Secret_Agent_Pkg;
```

3.6 Other Object-Oriented features

Ada has a number of other features that enhance the use of tagged types.

3.6.1 Package Ada.Tags

In some cases it's useful to deal directly with the tags associated with objects, such as for debugging purposes or for inputting and outputting tagged data to a data stream that requires a symbolic representation. The tag of a type or a class-wide object can be accessed by use of the attribute `Tag` (e.g., `Obj' Tag`). The predefined library package `Ada.Tags` provides operations to access the name of the type associated with an object's tag and to convert to and from the internal form of a tag (generally represented internally as an address) and the external string representation of the tag. It's also possible to access the tags of parent and interface types.

3.6.2 Ada.Tags.Generic_Dispatching_Constructor

It is sometimes necessary to create an object of a tagged type given a tag and some parameter values, for example in applications where tagged data is streamed in from an external device. A predefined generic unit, `Generic_Dispatching_Constructor`, which is a child unit of package `Ada.Tags`, provides this capability:

```
generic
  type T (<>) is abstract tagged limited private;
  type Parameters (<>) is limited private;
  with function Constructor (Params : not null access Parameters) return T
    is abstract;

function Ada.Tags.Generic_Dispatching_Constructor
  (The_Tag : Tag;
   Params  : not null access Parameters) return T'Class;
```

By instantiating with a parameter type and a constructor function returning any tagged type, the resulting function can be called with a tag and parameters to construct a value of the type associated with the tag.

3.6.3 Predefined Containers

The Ada predefined library provides a number of generic packages for *containers*, encapsulating various common data structure abstractions. These include linked lists, vectors, maps (hashed and ordered), and sets (hashed and ordered). In the upcoming Ada 2012 standard, additional container forms are being added for multiway trees, synchronized queues, and priority queues. For most of the containers, both bounded and indefinite forms are defined, supporting fixed-size and variable-sized component types respectively.

3.7 Support for Programming by Contract in Ada 2012

The upcoming Ada 2012 language revision includes several new features that facilitate debugging and program correctness analysis. These are defined as so-called *aspects*, which can be specified using a new syntax called *aspect specifications* that allow a variety of properties to be defined for types, objects, and subprograms. These properties are most commonly Boolean expressions, and Ada 2012 includes several new expression forms that will be convenient for contract-based programming.

Aspects are specified as part of the declaration of entities, such as object definitions, type declarations, all manner of subprogram declarations, component declarations and entry declarations. Specifiable aspects include among others size, alignment, packing, as well as the novel predicates described later. For example, the following example shows the definition of a class-wide type invariant, which applies to all extensions of the type:

```
type Shape is tagged private
  with Invariant'Class => Is_Valid (Shape);

function Is_Valid (Obj : Shape) return Boolean;
```

This invariant is checked on exit from any visible subprogram that is a primitive operation of the type.

Aspects may be statically verified at compile time or dynamically at execution time.

3.7.1 New Expression Forms.

Ada 2012 includes conditional expressions and case expressions, quantified expressions, and expression functions.

3.7.2 Conditional and Case expressions.

These are the obvious generalizations of the corresponding statement forms. These expressions must be parenthesized, or else appear within parentheses as an actual parameter in a call.

```
X := (if Z > 0 then 100 else 101);
```

```
Trigger := (if Enabled (B) then B > Unsafe);
```

```
Slot := (case X mod 4 is
         when 0 | 3 => 'a',
         when 1 => 'b',
         when 2 => 'c');
```

Note the shortcut in the second example: if the type of the expression is Boolean, a default value of True is assumed if there is no else part.

3.7.3 Quantified Expressions.

An Ada 2012 quantified expression corresponds exactly to its namesake in first-order logic. It indicates that a Boolean property holds for all or some elements of a collection, which may be an array or a container. For example, the assertion that an integer Num is prime can be written (although inefficiently) as follows:

```
(for all J in 2 .. Num / 2 => Num mod J /= 0)
```

Universally quantified expressions use the keyword **all**. For existentially quantified expressions Ada 2012 introduces a new reserved keyword **some**. The predicate indicating that a set Names_Table contains at least a non-null string can be written as follows:

```
(for some S of Names_Table => S'Length > 0)
```

This example uses another Ada 2012 syntactic innovation: an iteration over the elements of a container does not need an explicit cursor that denotes successive elements of the container.

3.7.4 Aspects.

An aspect may be specified for a subprogram, private type, subtype, and object, and is attached to the corresponding declaration. Aspects generalize attribute definition clauses and pragmas under a single syntax that uses the keyword **with** to introduce a comma-separated list of relevant aspects.

```
function Sqrt (X : Long_Float) return Long_Float
with Pre => X >= 0.0;
```

```
type Bit_Vector is array (positive range <>) of Boolean
with Pack => True;
```

3.7.5 Preconditions and Postconditions.

Preconditions and postconditions can be specified for subprograms by means of the `Pre` and `Post` aspects.

```
procedure Sort (Table : in out Vector)
with Post =>
  (for all J in Table'Range =>
```

```

J = Table'Last
  or else Table (J) <= Table (J + 1)

```

Preconditions and postconditions are boolean expressions so there can be several combined using logical operators.

3.7.6 Type Invariants.

Type invariants can be specified for private types. They are expressed by means of the aspects `Invariant` and `Invariant'Class`. Invariants are restricted to private types because one can then specify precisely the places at which invariants must be verified: the invariant expression associated with a type is evaluated as a postcondition of any operation that creates, evaluates or returns a value of the type. The aspect `Invariant'Class` given for a type `T` indicates that the given invariant is inherited by all types derived from `T`. This aspect can only be specified for a tagged type. If a type has both inherited and specified invariants, the resulting invariant is the conjunction of all of them.

```

generic
  type Item is private;
package Stacks is
  type Stack is private with Invariant => Is_Valid (Stack);

  procedure Push (S : in out Stack; I : in Item)
  with Pre => not Is_Full (S),
       Post => not Is_Empty (S);

  procedure Pop (S : in out Stack; I : out Item)
  with Pre => not Is_Empty (S),
       Post => not Is_Full (S);

  function Top (S : in Stack) return Item
  with Pre => not Is_Empty (S);

  Stack_Error : exception;

  function Is_Empty (S : in Stack) return Boolean;
  function Is_Full (S : in Stack) return Boolean;
  function Is_Valid (S : in Stack) return Boolean;

private
  -- not specified
end Stacks;

```

Note that the `Invariant` and `Pre/Post` aspects are calls to functions that have not been declared yet. An aspect is resolved at the end of the the current list of declarations, and not at the point where it appears. In this way aspects can mention auxiliary entities that must be declared after the type or subprogram itself.

VULNERABILITIES AND THEIR MITIGATION

This section discusses vulnerabilities resulting from the nature of object-oriented programming and the language features that support OOP.

4.1 Lack of Encapsulation

Unencapsulated data (i.e., data whose internal state can be referenced or assigned outside the code that requires such access) is problematic for several reasons. First, coding errors can cause data to have inconsistent state, resulting in vulnerabilities or hazards. The DO-178 standard specifically cites the corruption of global data as an error that requirements-based software integration testing should detect; using encapsulation to prevent such errors simplifies the testing and the related data coupling analysis. Second, modifications to the definition of the data structure during program maintenance may necessitate source code changes in code that accesses the data; i.e., lack of encapsulation makes module interfaces too broad. Encapsulation involves defining procedural interfaces to data and declaring the data items themselves such that their accessibility is restricted.

Ada addresses this issue via several language features, depending on whether the encapsulation is for data objects or data types. Somewhat simplified:

- The placement of an object's declaration within a package determines where the object's name is visible. The minimal encapsulation is to place the declaration in the visible part of a package spec; it is then visible to any "client" unit (i.e., any unit that "with"s the package). An intermediate level of encapsulation is to place the declaration in the private part of a package spec; it is then visible within the private part and body of child units, but not clients. For full encapsulation, the declaration may be placed in a package body. It is then visible only within that package body (and subunits), and not to child units or clients.
- A private type encapsulates the fields of a (tagged) record. The most common case is a (tagged) type declared as private in the visible part of a package spec, with the complete declaration of the type (as a record type) occurring in the private part of the package. A client unit cannot reference the fields of the record. The fields are accessible in the package body and in the implementation of child units (ie in the private parts and bodies of child packages).

Encapsulation does introduce several issues, however. One is performance related: accessing a field through a procedural interface is much more expensive in execution time than simply fetching the field directly. This problem is addressed by applying a compiler directive, `pragma Inline`, to the subprogram. The pragma advises the compiler to expand each invocation of the subprogram in a macro-like fashion, thus avoiding the linkage overhead of out-of-line calls.

A second issue involves the robustness of requirements-based testing. For safety certification standards such as DO-178, the developer needs to demonstrate that all low-level requirements are met. For requirements related to error handling, a test program may need to assign to a variable a value that could not occur in the course of normal processing. If the variable is encapsulated (e.g., declared in the private part of a package), there is no way to write a test as a "client" unit (one that "with"s the defining package). Ada offers a convenient solution to this problem: the test can be expressed as a child unit of the defining package, thus providing the necessary visibility onto the

variable. And instead of encapsulating data objects in package bodies, the programmer can place the declarations in the visible part of private children. This allows reuse of implementation, and again allows test programs to be written as children of the original package.

4.2 Dynamic Dispatch and Substitutability

The *Liskov Substitution Principle* (LSP) [LW94] states that within a class hierarchy, an instance of a subtype is also an instance of a supertype, and thus a subtype instance should be usable whenever a supertype instance is required.

Dynamic Dispatch represents a clear vulnerability when it is used within a class hierarchy that has not been built with the LSP. Adding a new class (and its methods) to a class hierarchy can affect any software component that dynamically dispatches to one of the methods defined in this hierarchy. Even if this component has been verified and tested with the previous state of the hierarchy, it can suddenly start failing because the new method does not fulfil the requirements or expectations of its ancestors. The failure can come from the fact that the newly added version of the method expects too much from its context (stronger precondition) or fails to provide some elements expected by the context as a result of the call (weaker postcondition). It is uncomfortable to be faced with situations where unit testing might be invalidated by changes that do not visibly affect the unit under test.

In the context of traditional imperative languages, it is common to base the verification of software components on tests derived from the requirements of the component. This is referred to as *requirements-based testing*. The quality of the testing is assessed, among other things, through coverage metrics. For instance, in the Avionics DO-178 standard, the requirements-based tests must cover all statements as soon as the safety level is C or above. When it comes to OO systems, we have to deal with a new kind of statement – a dispatching call – and we need to decide what it means to “cover” such a call. Does it mean that the thread of execution should go through this “statement” at least once as for any traditional statement, or should it cover all the possible targets of the dispatch?

To see the difference between the two approaches, we can simulate a dispatching call with a case statement. Consider the following example:

```
type T0 is tagged private;
procedure P (X : T);

type T1 is new T0 with private;
overriding procedure P (X : T1);

type T2 is new T0 with private;
overriding procedure P (X : T2);

...
Obj1, Obj2 : T0'Class := ...;
Obj1.P; -- dispatching call
...
Obj2.P; -- dispatching call
...
```

If we consider the previous dispatching calls as any other subprogram call, we could think that a simple test case executing the dispatching calls just once would fulfill the required statement coverage objectives.

However, dispatching calls are a different kind of statement, and the above dispatching calls can be considered conceptually equivalent to case statements such as:

```
Obj1, Obj2 : T0'Class := ...;
case Obj1'Tag is
  when T0'Tag => T0(Obj1).P; -- static calls
  when T1'Tag => T1(Obj1).P;
  when T2'Tag => T2(Obj1).P;
end case;
...
case Obj2'Tag is
  when T0'Tag => T0(Obj2).P; -- static calls
```

```

    when T1' Tag => T1 (Obj2) .P;
    when T2' Tag => T2 (Obj2) .P;
end case;
...

```

Such implicit case statements could also be moved out-of-line and still provide an equivalent effect:

```

Obj1, Obj2 : T'Class := ...;
...
Dispatch_To_P (Obj1);
...
Dispatch_To_P (Obj2);
...

procedure Dispatch_To_P (Obj : T'Class) is
begin
  case Obj' Tag is
    when T0' Tag => T0 (Obj) .P;  -- static calls
    when T1' Tag => T1 (Obj) .P;
    when T2' Tag => T2 (Obj) .P;
  end case;
end Dispatch_To_P;

```

Those two equivalent sequences of non-OO code have very different consequences in terms of statement coverage obligations. In the first case, each possible target for the dispatching operation should be exercised at each dispatch point if we want to achieve 100% statement coverage. This is called *pessimistic testing* and leads to exponential testing requirements which becomes impractical as soon as the class hierarchy becomes large. In the second case, each possible dispatch must be tested at least once among the various dispatch points leading to it. This is called *optimistic testing* and, in contrast with pessimistic testing, is not sufficient to address the vulnerability we are considering here : Errors related to this vulnerability would remain undetected if the faulty method (in the sense that it does not respect LSP) is only called, and thus covered, from innocuous dispatch points while other dispatch points exist and provide a context sensitive to such errors.

The OO supplement of DO-178C [RE2010] offers an original solution to this dilemma, which we recommend following. Instead of trying to adapt the definition of statement coverage to OO languages and in particular to dispatching calls, it suggests adding a specific verification activity dealing with this vulnerability. This activity is called *Local Type Consistency Verification* and ensures that LSP is sufficiently respected to address the vulnerability. Three possibilities are offered:

- Formally verify substitutability.
- Ensure that each tagged type passes all the tests of all its parent types which it can replace.
- For each dispatch point, test every method that can be invoked (pessimistic testing)

The *local* in “Local Type Consistency” refers to the fact that one can limit the consistency verification to the cases that actually occur in the application. For instance, when a method is never dynamically dispatched to, it is not necessary to verify that its precondition is no stronger than its parent’s.

The first possibility offers a direct verification of the LSP. It is the recommended path when appropriate formal methods can be used, since they provide the highest level of confidence that the objective is met. SPARK Pro has the capability of conducting such a formal proof of LSP. See section *Formal Proofs* below.

The second possibility is well adapted to verification based on unit testing. In such a context, each class’s method is associated with a set of tests that verify its requirements. An overridden method usually has extended requirements compared to the method it overrides, so it will be associated with an extended set of tests. Each class can be tested separately by calling the tests of its methods. In order to verify substitutability of a given tagged type by testing, the idea is to run all the tests for all parent types with objects of the given tagged 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. Aunit, the unit testing tool included in GNAT Pro, offers the necessary support for automating such verification activities and provide specific examples showing the detection of LSP violations.

The third possibility corresponds precisely to the pessimistic testing strategy described earlier. It may remain the simplest verification method when dispatching calls are rare and the class hierarchy is shallow and/or narrow. In

the context of GNAT Pro, the GNATstack tool can locate all the dispatching points of the application and identify the potential destination subprograms for each. See section *Using GNATstack for Stack Resource Analysis* below. Note that GNATstack locates dispatching calls and potential targets for each of them, but the coverage of these calls need to be verified to meet the requirements (coverage analysis is outside the scope of GNATstack).

4.3 Improper Overriding

The semantics of inheritance in Object-Oriented languages may result in two kinds of programming errors:

- Declaring a new or overloaded subprogram when overriding was intended
- Overriding a subprogram when declaring a new subprogram was intended

The first issue (sometimes known as *unintended non-overriding*) typically occurs when the subprogram name is misspelled. Here is an example in Ada:

```
package Text_Processing is
  type Text is tagged private;
  procedure Italicize ( Item : in out Text );
  ...
end Text_Processing;

with Text_Processing; use Text_Processing;
package Rich_Text_Processing is
  type Rich_Text is new Text with private;
  procedure Italicise ( Item : in out Rich_Text );
  ...
end Rich_Text_Processing;
```

This code is legal and will compile, but the `Italicise` procedure does not override the parent type's `Italicize`. Instead, the latter procedure is implicitly inherited by the `Rich_Text` type, which is probably not the programmer's intent. Since an overriding subprogram may perform subclass-specific safety or security checks, the invocation of the parent subprogram on a subclass instance can introduce a vulnerability.

The second issue (*unintended overriding*) can arise when a new subprogram is added to a parent tagged type during program maintenance, after an inheritance hierarchy already exists. For example:

```
package A is
  type T is tagged private;

  procedure P (Item : in out T);
  ...
end A;

with A; use A;
package B is
  type T1 is new T with private;

  -- Inherit P
  procedure Q (Item : in out T1); -- New subprogram
  ...
end B;
```

Suppose that during program maintenance the author of package A (who might not even be aware of package B) modifies the package to introduce a procedure Q:

```
package A is
  type T is tagged private;

  procedure P (Item : in out T);
  procedure Q (Item : in out T); -- New subprogram
  ...
end A;
```

When B is recompiled, its declaration of Q is interpreted as an overriding of A.Q. Therefore an invocation of A.Q on an object of type A.T'Class that is currently a B.T1 will dynamically bind to the originally declared subprogram B.Q. This scenario was not anticipated by the author of B. Mistakenly invoking one subprogram instead of another can have unpredictable effects, so this sort of error may introduce a vulnerability.

Ada 2005 has addressed both issues through a feature that allows the programmer to explicitly mark a primitive subprogram as *overriding* or *not overriding*. If a subprogram is intended as an overriding of a subprogram from a parent type, then it may be declared with the **overriding** reserved word. In such cases it is a compile-time error if the subprogram is not such an overriding. Likewise, if a subprogram is declared as **not overriding**, then it is a compile-time error if the declaration overrides a subprogram declaration from the parent type.

Reformulating the unintended non-overriding example with the Ada 2005 syntax:

```
package Rich_Text_Processing is
  type Rich_Text is new Text with private;
  overriding procedure Italicise ( Item : in out Rich_Text );
  ...
end Rich_Text_Processing;
```

The compile-time error for *Italicise* will alert the programmer to the need to correct the spelling.

In the unintended overriding example:

```
package B is
  type T1 is new T with private;

  -- Inherit P
  not overriding procedure Q(Item : in out T1);
  ...
end B;
```

The compile-time error for Q's declaration (after the A package has introduced Q for type T) will alert the programmer to the need to change the subprogram's name. This might have a ripple effect (it is an example of the *fragile base class* problem in OOP, where a change to a base class may induce a cascade of source code changes in subclasses), but this is better than having an incorrect version of the subprogram invoked at run time.

For upwards compatibility with Ada 95, the inclusion of **overriding** or **not overriding** for a derived tagged type's primitive subprograms is optional in Ada 2005. The use of **overriding** is strongly encouraged and can be enforced by an appropriate coding standard rule.

However, the case for explicitly specifying **not overriding** is weaker:

- The syntax is rather heavy for what will be a common scenario
- A program that compiles correctly with *overriding* and *not overriding* declarations is not necessarily free of "unintended non-overriding" errors. In particular, a subprogram with neither of these designations might or might not be an overriding of an inherited subprogram. What is really needed is a check that *overriding* is supplied for each overriding declaration, and only for such declarations. There is no need to supply *not overriding*.
- The error that *not overriding* is designed to prevent – the unintended overriding of a subprogram that was added to a parent type – will be caught if the user supplies two switches to the GNAT Pro compiler:
 - `-gnaty0`, which warns about overriding declarations that are not marked with the *overriding* keyword
 - `-gnatwe`, which treats warnings as errors

As a result of these considerations, the recommendation is to not use **not overriding**, and instead to ensure that all overriding declarations (for a derived tagged type's primitive subprograms) are marked as *overriding*. This can be enforced through the switches mentioned above or through a coding standard rule (which is implemented through these switches). If *overriding* is used systematically then *not overriding* is useless.

4.4 Multiple Inheritance

The major vulnerabilities associated with multiple inheritance are described in the (Draft) *Object-Oriented and Related Technology Supplement* [RE2010] to the upcoming DO-178C standard. In summary:

- With multiple interface inheritance, if two interfaces contain an identical method signature, a (non-abstract) class that inherits from these interfaces will need to provide a single implementation for both. However, the intent (contracts) of the methods in the two interfaces may be completely unrelated, raising the issue of how one implementation can work for two different purposes.
- With multiple implementation inheritance, name clashes and “diamond inheritance” problems (described earlier under *Inheritance* in *Object-Oriented Programming (OOP)*) complicate traceability analysis and software verification.

In Ada, the “multiple interfaces with common method signature” problem can be solved by adding a discriminant to the derived type, which can reflect which interface is intended. Here is an example:

```

package I1_Pkg is
  type I1 is interface;
  type I1_Class_Ref is access all I1'Class;
  procedure P(X:I1) is abstract;
  procedure Q1(X:I1) is abstract;
end I1_Pkg;

package I2_Pkg is
  type I2 is interface;
  type I2_Class_Ref is access all I2'Class;
  procedure P(X:I2) is abstract;
  procedure Q2(X:I2) is abstract;
end I2_Pkg;

with I1_Pkg, I2_Pkg;
use I1_Pkg, I2_Pkg;
package T_Pkg is
  type Selector is (I1_Value, I2_Value);
  type T(S : Selector) is new I1 and I2 with
    record
      A, B : Integer;
    end record;
  type T_Class_Ref is access all T'Class;
  procedure P(X:T); -- Implements I1.P and I2.P
  procedure Q1(X:T);
  procedure Q2(X:T);
end T_Pkg;

package body T_Pkg is
  procedure P(X:T) is
  begin
    case X.S is
      when I1_Value => ... -- (1)
      when I2_Value => ... -- (2)
    end case;
  end P;

  procedure Q1(X:T) is ...;
  procedure Q2(X:T) is ...;
end T_Pkg;

with I1_Pkg, I2_Pkg, T_Pkg;
use I1_Pkg, I2_Pkg, T_Pkg;
procedure Test is
  Ref1 : I1_Class_Ref := new T(I1_Value);
  Ref2 : I2_Class_Ref := new T(I2_Value);

```

```

begin
  T(Ref1.all) := (I1_Value, 10, 20);
  T(Ref2.all) := (I2_Value, 100, 200);
  P(Ref1.all); -- Branch (1)
  P(Ref2.all); -- Branch (2)
end Test;

```

The issues associated with multiple implementation inheritance do not arise in Ada, since Ada does not supply this language feature. However, even without an explicit language feature, Ada is able to obtain the main benefits of multiple implementation inheritance. The most common case in practice is the so-called *mix-in*, where a subclass needs to inherit the implementation of methods from one main superclass, but only the methods signatures from other classes. This exactly corresponds to the multiple interface inheritance facility that Ada provides. And in those rare cases where multiple implementation inheritance (say from T1 and T2) is needed, it can be simulated through *delegation*. Declare a type T3 that derives from one of these types (say T1), and in the record extension for T3 define a component with type T2. For each primitive subprogram P for T2, T3 will declare a corresponding subprogram P that either directly calls P on the view conversion to T2 (simulating inheritance), or else declares a new subprogram P with the appropriate effect (simulating an overriding).

An example of *mix-in* with a subclass inheriting the implementation of methods from one main superclass, and the methods signatures from another class (interface) is the following. There is a class Shape which represents colorless shapes, and we add the interface Colorful:

```

type Shape is tagged null record;

type Colorful is interface;

procedure Set_Color (Obj : in out Colorful; Color : Color_Type) is abstract;
function Get_Color (Obj : Colorful) return Color_Type is abstract;

generic
  type T is tagged private;
package Make_Colorful is
  type Colorful_T is new T and Colorful with private;
  procedure Set_Color (Obj : in out Colorful_T; Color : Color_Type);
  function Get_Color (Obj : Colorful_T) return Color_Type;
private
  type Colorful_T is new T and Colorful with
    record
      Color : Color_Type;
    end record;
end Make_Colorful;

package Colorful_Shape is new Make_Colorful (Shape);

```

4.5 Static Binding and Overriding

As mentioned earlier, static binding corresponds to a direct call to a method. Overriding implies the same method is implemented in different ways for different elements of a class hierarchy. Using both at the same time opens a vulnerability when an object's static compile-time known type differs from its dynamic (run-time) type because such an object can end up being manipulated by a method implementation that does not match its actual type but one of its parent types (superclasses).

Ada is a strongly-typed language. Furthermore, it differentiates class-wide types from specific types, so one could have expected the mismatch between dynamic and static type to be rare or impossible. This is not the case, however, because the language supports view conversions both explicit, through type conversions to a parent type, and implicit, through method inheritance as shown in the following example. The vulnerability appears when manipulating an object with its specific methods is essential to the consistency of the object.

```

-- parent class
type Dated_Thing is tagged private;

```

```
-- methods
procedure Set_Date (Obj : in out Dated_Thing; D : Date);
function Get_Date (Obj : in Dated_Thing) return Date;
procedure Copy_Date (From : in Dated_Thing; To : in out Dated_Thing);

-- subclass
type Logged_Thing is new Dated_Thing with private;

-- one method is overridden, the other two are inherited
overriding procedure Set_Date (Obj: in out Logged_Thing);
-- change the date but also keep track of the change in a log

-- first example of vulnerability (explicit view conversion)
procedure Do_Something (X : in out Dated_Thing) is
begin
    ...
    X.Set_Date (My_Date);
    ...
end Do_Something;

My_Thing : Logged_Thing;
...
    Do_Something (Dated_Thing (My_Thing));
    -- the date will be set but not logged
...

-- second example (implicit view conversion)
-- this is the implementation of Dated_Thing's third method
procedure Copy_Date (From : in Dated_Thing; To : in out Dated_Thing) is
begin
    To.Set_Date (From.Get_Date);
end Copy_Date;
-- implementation is straightforward and seem to apply as is for
-- Logged_Thing so it can be inherited.
My_Thing1, My_Thing2 : Logged_Thing;
...
    Copy_Date (My_Thing1, My_Thing2);
    -- the date will be changed but not logged
...
```

When detected, those vulnerabilities can easily be countered: in the first example, the `Do_Something` routine could have taken a class-wide parameter, that would have made the type conversion on the caller side unnecessary and even if it was kept, it would not matter because the call to `Set_Date` in its body would have been dispatching in any case. In the second example, the problem can be addressed by simply copy/pasting the `Copy_Date` routine instead of inheriting it. The problem is not really how to address such issues when they are discovered but rather make sure that the problem cannot happen thanks to an appropriate coding standard. The Object-Oriented Technology in Aviation handbook [OOTiA] describes this vulnerability and suggests a rule to address it called *the simple dispatch rule*. This rule specifies that any call to a method should be dispatching except in the very specific case of a method calling its immediate parent method. There is a GNATcheck rule (see *Writing a Coding Standard with GNATcheck*) `Direct_Calls_To_Primitives` which prevents any static dispatching.

4.6 Memory Management Issues

A polymorphic variable can denote objects from different classes at different times. Since a class may be added to an inheritance hierarchy after the code containing the declaration of a polymorphic variable has been compiled, the compiler cannot predict in advance the maximum size for the object that the variable denotes. As a result, polymorphic variables are implemented as references (pointers) to their denoted objects, or, in Ada parlance, as access values that designate their associated objects.

The representation of polymorphic variables as access values raises several issues:

1. Is dynamic allocation required?
2. If so, then various vulnerabilities may arise, from both the application code and the implementation of the compiler vendor's dynamic memory management run-time library. How are these vulnerabilities mitigated?

These issues will be discussed in the next two sections.

4.6.1 Avoiding Dynamic Allocation

Dynamic allocation is not intrinsic to OOP in Ada. In simple Object-Oriented applications the programmer can declare the needed objects as aliased variables (perhaps pooled into arrays) in a library-level package, and then obtain references to them via the `'Access` attribute. The following fragment uses the examples shown earlier:

```
with Text_Processing, Rich_Text_Processing;
use Text_Processing, Rich_Text_Processing;
package Objects is
  Max_Texts      : constant := ...;
  Max_Rich_Texts : constant := ...;
  Texts          : array (1..Max_Texts) of aliased Text;
  Rich_Texts     : array (1..Max_Rich_Texts) of aliased Rich_Text;
end Objects;

with Text_Processing, Rich_Text_Processing, Objects;
use Text_Processing, Rich_Text_Processing, Objects;
procedure Example is
  type Text_Class_Ref is access all Text'Class;
  Ref : Text_Class_Ref;
begin
  ...
  Ref := Texts(1)'Access;
  Italicize (Ref.all); -- Text_Processing.Italicize
  ...
  Ref := Rich_Texts(1)'Access;
  Italicize (Ref.all); -- Rich_Text_Processing.Italicize
  ...
end Example;
```

Thus the full range of OOP features may be used – inheritance, class-wide types, polymorphism, dynamic binding – without requiring dynamic allocation (neither implicit nor explicit).

The absence of dynamic allocation can be enforced by supplying two pragmas:

- `pragma Restrictions (No_Allocators)`, which prohibits all explicit allocators, and
- `pragma Restrictions (No_Implicit_Heap_Allocations)`, which prohibits the use of implicit allocations for purposes such as array descriptors.

Independent of OOP, other uses of dynamic allocation arise when a data structure needs a flexible representation so that it can grow or shrink. In some situations the maximum size is unknown; for such unbounded data structures it will be necessary to use dynamic allocation. However, in many cases a maximum size can be predicted. For such bounded data structures the programmer can use discriminated record types or, for better encapsulation, one of the new Ada 2012 bounded container types such as `Bounded_Doubly_Linked_Lists`. The GNAT Pro implementation of discriminated record types and the bounded container types does not use dynamic allocation.

4.6.2 Using Dynamic Allocation Safely

Although some OOP scenarios can be modeled with a statically-determined set of objects, it is more typical to require additional flexibility that implies dynamic allocation. The issue is how to do this without introducing vulnerabilities.

From an application programmer's perspective, there are several main requirements:

1. Ensure that no allocation request fails

2. If the language provides an explicit deallocation construct, ensure that no object is deallocated while it is still accessible to the program (i.e., avoid *dangling references*).
3. If the implementation provides storage reclamation (i.e., garbage collection), ensure that the garbage collector's execution does not cause real-time deadlines to be missed

Since Ada provides explicit deallocation – the generic procedure `Ada.Unchecked_Deallocation` – rather than assuming a garbage collector, the third issue does not arise.

There are several approaches to meeting the first two requirements. A simple solution is to restrict dynamic allocation to the package elaboration phase of program execution, and to forbid unchecked deallocation. A typical style is to allocate pools of objects, with program control over when they are no longer needed (so that they may be reused). There are no problems concerning heap exhaustion or dangling references, but in a DO-178 context the developer will need to verify the object pool management via appropriate analysis.

The restriction of dynamic allocation to package elaboration, and the prohibition of unchecked deallocation, can be enforced by supplying two pragmas:

- `pragma Restrictions (No_Local_Allocators)`
- `pragma Restrictions (No_Unchecked_Deallocation)`

If these restrictions are too constraining, then a range of more general strategies are available. One is to allow namespaces (subprograms, blocks) to perform dynamic allocations in a locally created pool of sufficient size, to ensure that references to such objects are never copied to more global variables, and to deallocate the entire pool at scope exit. The developer would need to demonstrate that requirements 1 and 2 are met (i.e. that the pool is large enough, and then references do not “escape”). With any additional generality regarding the usage of allocation and deallocation, the price is additional complexity in certification.

An example of how to define a storage pool and use it for class-wide types is the following. Users can define the way dynamic memory from that pool can be allocated and deallocated:

```
type My_Storage_Pool_Type is new
  System.Storage_Pools.Root_Storage_Pool with null record;

overriding function Storage_Size
  (Pool : My_Storage_Pool_Type)
  return System.Storage_Elements.Storage_Count;

overriding procedure Allocate
  (Pool      : in out My_Storage_Pool_Type;
   Address   : out System.Address;
   Storage_Size : System.Storage_Elements.Storage_Count;
   Alignment  : System.Storage_Elements.Storage_Count);

overriding procedure Deallocate
  (Pool      : in out My_Storage_Pool_Type;
   Address   : System.Address;
   Storage_Size : System.Storage_Elements.Storage_Count;
   Alignment  : System.Storage_Elements.Storage_Count);

My_Pool : My_Storage_Pool_Type;

type Shape is tagged null record;
type Any_Shape is access Shape'Class
  with Storage_Pool => My_Pool;
```

A related requirement when using access types is to ensure that no attempt is made to dereference (apply `.all` to) a `null` access value. This is a data flow analysis issue, which can be facilitated through static analysis (for example via CodePeer Pro).

COMPLEXITY MANAGEMENT

5.1 Control Inheritance Depth / Width and Multiple Inheritance

Although inheritance is intrinsic to OOP, a class hierarchy that is not properly designed may complicate system maintenance and increase the effort in safety certification. The heart of the problem is what is sometimes referred to as the *fragile base class* issue. A change to a base class in an inheritance hierarchy may have a ripple effect that requires inspection, modification, recompilation, and retesting/reverification of all classes in the hierarchy. Other sections of this document deal with some of the mitigation strategies: choosing appropriate data encapsulation and separate compilation relationships is discussed in *Control Class Coupling*, and avoiding unintentional overriding is discussed in *Improper Overriding*. This section addresses complexity issues associated with the depth and width of the inheritance hierarchy, and with multiple inheritance.

5.1.1 Deep hierarchies

Deep inheritance hierarchies raise several issues:

- Such hierarchies may be symptoms of poor design. As a contrived example, although it is possible to define a `Stack` base class with a single method `Push`, and then a subclass `PoppableStack` with an additional method `Pop`, and then a further subclass `QueryablePoppableStack` with an additional method `Size`, such a class hierarchy would of course be highly suspect. A class designer needs to provide a complete and coherent base class from the start, rather than rely on subclassing to compensate for errors of omission.
- It is more difficult for the human developer to understand and maintain the class structure. Deep inheritance hierarchies require the reader to refer back to multiple superclasses in order to understand the behavior of a “leaf” subclass.

The threshold for “too deep” is inexact, but beyond around 4 or 5 levels the complexity accelerates. To specify a limit on inheritance depth, the developer can use the GNATcheck tool with the rule `Deep_Inheritance_Hierarchies`, using the maximum inheritance depth as parameter of the rule.

There is also a design/implementation technique for reducing inheritance depth: use variant records instead of inheritance to model variation among class instances. Although the variant record style is generally regarded with suspicion in the OO community, there are circumstances when it may provide the simplest solution. In Ada you can declare a discriminated tagged type where the discriminant establishes the variant that is present, so the OO and variant record styles can be combined.

5.1.2 Wide hierarchies

Wide inheritance hierarchies are not necessarily as problematic as deep ones. Different subclasses of the same base class may be completely independent and may participate in separate subsystems, so the number of such subclasses does not raise complexity issues. The developer or maintainer of each subsystem does not need to be aware of the sibling classes used in other subsystems.

On the other hand if the developer or maintainer needs to understand a software architecture with one superclass and a large number of immediate subclasses then that could present problems, especially if there is coupling

among the subclasses. If several of the subclasses share some common traits then it may make sense to merge these classes into a single class, perhaps abstract, which is then subclassed (possibly with interface inheritance) to obtain the necessary variation.

There is no GNATcheck rule to automatically verify this property, and it needs to be verified by manual inspection of the code.

5.1.3 Multiple inheritance

The problems with multiple implementation inheritance are covered elsewhere in this document. Since Ada only supplies interface inheritance, it avoids these issues. There are, however, a few points that need to be considered with multiple interface inheritance.

- *Inheriting the same subprogram signature from multiple interfaces.* This issue is discussed in section [Multiple Inheritance](#).
- *Inheriting conflicting subprogram signatures.* The following example illustrates the problem:

```
type I1 is interface;
procedure P(I : I1; N : Natural) is abstract;
procedure Q1(I : I1) is abstract;

type I2 is interface;
procedure P(I : I2; N : Integer) is abstract;
procedure Q2(I : I2) is abstract;
```

It is impossible to declare a type that inherits from both I1 and I2, since there is no way to declare P. (Ada's derivation / overriding rules require subtype conformance, and the formal parameter cannot match both Natural and Integer.) One solution is to adopt a style in which base types rather than subtypes are used for formal parameters and function results of subprograms declared in interfaces.

- *Inheriting too many interfaces.* Similar to the issues raised with deep or wide hierarchies, inheriting from a large number of interfaces can interfere with program understanding. To specify a limit on the number of interfaces that a type inherits from, the developer can use the GNATcheck tool with the rule Too_Many_Parents, using the maximum number of parents as parameter of the rule.

5.2 Control Class Coupling

This section describes both the general issues associated with inter-module coupling and the specific issues that arise with class coupling in OOP. It shows how to mitigate these issues using Ada and GNAT Pro tools.

A major principle of encapsulation is to define the interface for a module to be as narrow as possible, in order to minimize the impact on other units when the module is changed. (The term *interface* is used here in the general software architecture sense: the properties of a module that may be assumed by external components. It is not referring to the Ada interface type facility.)

The “impact on another unit when the module is changed” is referred to as the *coupling* between the module and the other unit, and there are three principal possibilities:

- *No coupling.* No effect on other units (i.e., no need for either source code changes or recompilation)
- *Weak coupling / recompilation required.* Other units need to be recompiled but do not require source code changes (there is no possibility of compilation errors)
- *Strong coupling / source changes possible.* The other units' source code may need to be changed and will need to be recompiled

A significant vulnerability in some languages is to allow an executable program to be built without enforcing the needed recompilations. Commonly referred to as *version skew*, this situation can arise if a module is changed (for example so that a subprogram has a different signature) and recompiled, but some unit dependent on the old version (e.g., invoking the subprogram) is not recompiled. The resulting program will have unpredictable behavior. This sort of error is prevented in Ada.

Strong coupling introduces a number of problems. It implies a potential for complex inter-module relationships, and a resulting increase in effort in demonstrating compliance with safety certification standards such as DO-178B (for example, to show that data structures cannot be corrupted). Ada's package structure allows the package developer to control the trade-off between minimizing the package's coupling and maximizing its potential for reuse, based on where declarations appear. In the other direction, the developer of a unit that needs to reference a package's entities can, by choosing the separate compilation relationship between the unit and the package, determine the degree of coupling between the two. The major separate compilation relationships in Ada are the *client* relationship (if `with P` is in a context clause for `Q` then `Q` is said to be a client of `P`) and the *child* relationship (between a child unit and its parent package).

From the perspective of the package developer:

- The visible part of a package specification has a “source changes possible” coupling with all sections of each of its client units and each of its child units
- The private part of the specification for a package `P` has:
 - A “recompilation required” coupling with each of `P`'s client units
 - A “source changes possible” coupling with the complete specification of each of `P`'s private child units, and with the private part and body of each of `P`'s public child units
- The body for a target package has a “source changes possible” coupling with all subunits (and their dependents), and a “no coupling” relationship with all other units that depend on the package specification

More simply, a declaration in the visible part of a package specification has maximal reuse but also strongest coupling, a declaration in a package body has minimal reuse but also weakest coupling, and a declaration in the private part of a package specification has intermediate reuse (visibility in child units but not to clients) and intermediate coupling.

The above rules are qualified by the disclaimer that an implementation may optimize by avoiding recompilations that are unnecessary, and on the other hand may also add further dependencies in an implementation-defined manner. A common additional dependence occurs when a package body contains an inlined subprogram or the body of a generic unit; in such situations there may be a “recompilation required” coupling with units that invoke the subprogram / instantiate the generic.

Given the existence of a package `P`, the developer of a unit `U` that needs to reference an entity in `P` can choose between two principal separate compilation relationships:

- `U` may be a client of `P`
- `U` may be a child of `P` (i.e., `P . U`)

To minimize coupling, `U` should be a client of `P` unless `U` requires access to the declarations in the private part of `P`. As a client unit, `U` will not need to be changed if the private part or body of `P` is modified. On the other hand, if `U` does need to reference the private part of `P`, then `U` should be declared as a child of `P`. Generally, it will be a public child; in some circumstances (for example the encapsulation of implementation-oriented declarations) it will make sense to define `U` as a private child.

Several other aspects of Ada's separate compilation facility are relevant to the issue of coupling control:

- *with* clauses in Ada are not transitive. That is, if `R withs Q` and `Q withs P`, then `R` does not automatically gain access to the declarations in `P`'s visible part. That can help avoid propagation of the need to change source code when the visible part of `P` is modified.
- The nature of a unit `U`'s coupling with other units is always apparent at the beginning of the unit, from its *with* clause(s) and from its name (ie whether it is a child). This helps program readability.
- Ada 2005 introduced the concept of **limited with**, which allows different library packages to declare interdependent types. The *limited with* facility is an extremely weak coupling between two packages, since there are minimal assumptions that the packages can make about each other.

This section's discussion up to this point applies to Ada development in general. In the case of OOP, it is useful to distinguish two kinds of class coupling relationships: inheritance, and the usage of one class by another.

The decision with inheritance is whether to make the subclass's package a client or a child of the superclass's package. The recommendation above on minimizing coupling applies here. That is, if package `P` declares a

tagged type `T`, and package `U` declares a tagged type `T1` derived from `T`, and the bodies of the subprograms for `T1` do not need to access the representation of `T`, then `U` should be a client of `P`; it does not need to be a child. On the other hand, the use of child packages will be typical when inheriting from a tagged type that is either private or has a private extension, since defining the derived type's package as a client will not provide the necessary visibility. Thus an inheritance hierarchy of private tagged types will be modeled by a tree of child packages. The drawback to this organization is the *fragile base class* problem mentioned above in section *Control Inheritance Depth / Width and Multiple Inheritance*: if the root package is volatile (i.e., is undergoing frequent changes), then the source code in the child packages might need to be modified in response to the changes. This issue is inherent in OOP and is not specific to Ada.

In addition to inheritance, another form of class coupling is simply the usage of one class's entities by another; for example, class `C1` defining a field that has class `C2` or invoking a method from `C2`. In general, any given class will make use of entities from multiple other classes, and those dependencies can interfere with reuse and understandability, especially when there are a large number of dependencies or mutual dependencies among the classes. The GNAT Pro tool GNATmetrics includes switches that result in the output of various coupling complexity measures for an OO program:

- Each class's *efferent coupling*: the number of other classes that the given class depends on
- Each class's *afferent coupling*: the number of other classes that depend on the given class

There are also switches for deriving the efferent and afferent couplings of groups of classes. The report produced by GNATmetrics can help the developer assess the potential for problems associated with complex class couplings.

This section has thus far treated the concept of coupling in a syntactic sense: by choosing where to place a declaration, and whether to make a dependent a client or a child, the programmer controls which entities may be referenced, and from which kinds of units. However, and importantly, there is also a semantic sense to coupling. Although the interface between a package and a client unit is narrow syntactically, the client may be assuming certain properties of the behavior of the `with`'ed package. If so, then changes to the package body may cause the client unit to execute incorrectly. One such property is whether the invoked subprogram makes use of certain run-time facilities such as heap allocation or exception propagation. This can be detected through the use of pragma `Restrictions` or coding standard rules (GNATcheck). Another such property is execution time; in a real-time application the worst-case execution time of a subprogram is in fact part of that subprogram's semantic interface. Demonstrating that a subprogram invocation does not exceed its stated worst-case execution time requires verification activities separate from what is automatically provided by the compiler.

5.3 Minimize Need for Ada Run-Time Support and Facilitate Source-to-Object Traceability

Like most modern OO languages, Ada is an extensive language designed to cover the needs of a broad range of applications. When a constrained Ada subset is required in order to reduce costs and risks in meeting safety certification standards, Ada provides a mechanism for defining profiles [Ada2005], Section D.13. A profile defines a set of configuration pragmas which restrict the availability of language features and minimize the need for Ada run-time support. Ada 2005 [Ada2005] defines one profile (the Ravenscar profile) and implementations are allowed to provide other profiles. The GNAT Pro High-Integrity Edition [GNATHIE] provides four profiles: the Zero Footprint profile, the Cert Profile, the Ravenscar Profile, and the Full-Runtime profile). It also allows programmers to define their own profiles. Although limited in terms of dynamic Ada semantics, the profiles fully support static Ada constructs such as generic templates and child units.

In the area of Object-Oriented Programming, the following features can be restricted to minimize the Ada run-time support and facilitate source-to-object traceability:

- Tagged types defined at nested scopes, since their definition involves specific compiler support to handle pointers to primitives defined in nested scopes (for example, trampolines), and run-time support to serialize their elaboration (see [GNATHIE] Section 6.3).
- Controlled types, since their support requires extensive run-time support.
- Streams, since no run-time support is then needed to handle dispatching stream operations.
- Package `Ada.Tags`, since the full functionality of this package may not be needed.

- Class-wide types, if the safe use of record extensions is the unique OO requirement of the High-Integrity application. Their use can be limited through the Ada restriction identifier `No_Dispatch`.
- Allocation and unchecked deallocation of objects may be limited through restriction identifiers or under control of the High-Integrity application.

SAFETY AND VERIFICATION CONSIDERATIONS

6.1 Use of Pre/Post/Invariant Aspects for Verification

These new Ada 2012 aspects offer a new type of user-controlled checks that fit perfectly with OO and thus provide an extension to the safety brought by language defined checks.

Precondition checks, in particular, are particularly useful during integration testing and help with the DO-178 Control Coupling analysis.

Design by Contract [M97] (“DbC”) is a recognized way of building reliable object-oriented software. It forces the designer of a program to answer three questions for each class/method:

- What does the method expect?
- What does the method guarantee?
- What does the class maintain?

The answer to the first question is known as the method’s *precondition*, which is implemented in Ada 2012 as the `Pre` aspect. The answer to the second question is known as the method’s *postcondition*, which is implemented in Ada 2012 as the `Post` aspect. The answer to the third question is known as the class’s *invariant*, which is implemented in Ada 2012 as the `Invariant` aspect.

6.1.1 Preconditions and Postconditions

The precondition is a Boolean expression over formal parameters and global variables that is evaluated at subprogram entry. The postcondition is a Boolean expression relating formal parameters and global variables at subprogram entry and subprogram exit. This expression is evaluated at subprogram exit, and a special Ada attribute `'Old` makes it possible to refer to values at subprogram entry: `X'Old` refers to the value of `X` at subprogram entry. The value returned by a function is also available through attribute `'Result`: `F'Result` refers to the result of the current function `F`. Returning to the `Stack` example seen previously, we can express a richer contract on `Push`:

```
generic
  type Item is private;
  Max : Positive;
package Stacks is

  type Stack is private;

  function Is_Empty (S : Stack) return Boolean;
  function Is_Full (S : Stack) return Boolean;

  function Size (S : Stack) return Natural
    with Post => (if Is_Empty (S) then Size'Result = 0
```

```

        elsif Is_Full (S) then Size'Result = Max
        else Size'Result in 1 .. Max - 1);

    procedure Push (S : in out Stack; I : in Item)
        with Pre => not Is_Full (S),
             Post => Size (S) = Size (S)'Old + 1;

private
    ...
end Stacks;

```

Note that the attribute 'Old can be applied to any name, including a function call. Here, the expression `Size (S)'Old` might or might not be equivalent to `Size (S'Old)`, depending on the implementation of stacks, because the former refers to a call to `Size` at function entry (which is what we want), while the latter would refer to a call to `Size` at function exit (possible in a different heap).

If `Stack` is a tagged type, we can rewrite calls in the usual prefixed notation:

```

generic
    type Item is private;
    Max : Positive;
package Stacks is

    type Stack is tagged private;

    function Is_Empty (S : Stack) return Boolean;
    function Is_Full (S : Stack) return Boolean;

    function Size (S : Stack) return Natural
        with Post => (if S.Is_Empty then Size'Result = 0
                    elsif S.Is_Full then Size'Result = Max
                    else Size'Result in 1 .. Max - 1);

    procedure Push (S : in out Stack; I : in Item)
        with Pre => not S.Is_Full,
             Post => S.Size = S.Size'Old + 1;

private
    ...
end Stacks;

```

When compiled with assertions on (`-gnata` in GNAT), each precondition is checked before calling the subprogram, and each postcondition is checked before returning from the subprogram. Any failure to respect the contract expressed in the pre- or postcondition leads to raising an exception.

What is checked in preconditions and postconditions depends on the project/team objectives, but as a general guideline a contract should be a simple expression of rich constraints between the caller and the called subprogram. Typically, this is obtained by calling subprograms in the contract, in order to abstract operations. This is necessary both for encapsulation and for simplicity.

6.1.2 Control Coupling

According to DO-178, control coupling is “the manner or degree by which one software component influences the execution of another software component”. A large part of this influence is precisely captured by component subprograms’ preconditions. Verifying that preconditions are respected is therefore an effective means of controlling software components’ control coupling. It can be done either statically as described in a subsequent section or by enabling dynamic precondition checks during integration testing. The latter is facilitated in GNAT by the use of the following pragma:

```
pragma Check_Policy (Preconditions, On);
```

6.1.3 Invariants

An invariant is a property that all objects of a private type should respect, except when being modified by a primitive operation of the type. Typically, an invariant is not true during object construction/update/finalization occurring inside a primitive operation of the type.

In the stack example, an invariant can express that the stack is valid, with a definition of validity based on the internal representation of the stack. For example, `Is_Valid` could express that unused elements have a special value.

```
generic
  type Item is private;
  Unused : Item;
  Max    : Positive;
package Stacks is

  type Stack is private with Invariant => Is_Valid (Stack);

  function Is_Valid (S : Stack) return Boolean;

private

  type Items is array (1 .. Max) of Item;
  type Stack is tagged record
    Top  : Natural := 0;
    Data : Items;
  end record;

  function Is_Valid (S : Stack) return Boolean is
    (for all J in S.Top + 1 .. Max => S.Data (J) = Unused);

end Stacks;
```

All primitive operations should maintain the invariant of the object on which they operate. Ada 2012 only mandates that, after calling a subprogram that has a result or an **out** or **in out** parameter of the type, the invariant of this object is checked. These checks do not guarantee that the invariant is never broken, since the user may reference the value of a global variable in the invariant, and modify this global variable outside of the operations on the type. Rather, the invariant is a rich way to partially check such properties, as the user only has to state once a property checked in multiple places.

6.1.4 Inheritance

In some cases, one wants to make sure that the overriding operations of a derived type obey the restrictions imposed on the overridden operations. This is mandated in particular by the Liskov Substitutability Principle [LW94]. Then, one can use the inheritable versions of attributes `Pre`, `Post` and `Invariant`, called respectively `Pre'Class`, `Post'Class` and `Invariant'Class`.

Inherited preconditions are *ored* together to form the actual precondition. Inherited postconditions are *anded* together to form the actual postcondition. Inherited invariants are *anded* together to form the actual invariant.

If we consider the following code excerpt:

```
type Buffer is tagged private
  with Invariant'Class => Is_Valid (Buffer);

function Is_Valid (B : Buffer) return Boolean;
function Is_Full  (B : Buffer) return Boolean;
function Is_Empty (B : Buffer) return Boolean;

procedure Insert (B : in out Buffer; Item : Natural)
  with Pre'Class  => not Is_Full (B),
       Post'Class => not Is_Empty (B);
```

```
procedure Extract (B : in out Buffer; Item : out Natural)
  with Pre'Class => not Is_Empty (B),
       Post'Class => not Is_Full (B);

type Ordered_Buffer is new Buffer with private
  with Invariant'Class => Is_Ordered (Ordered_Buffer);

function Is_Ordered (B : Ordered_Buffer) return Boolean;
```

The preconditions, postconditions and invariants defined for class `Buffer` are inherited by class `Ordered_Buffer`. Class `Ordered_Buffer` defines another invariant, which is hence *anded* together with the inherited invariant, so both `Is_Valid` and `Is_Ordered` are invariants of class `Ordered_Buffer`.

6.2 Robustness of Dynamic Dispatch Mechanism

Dynamic dispatch through indirection is the technique commonly used by compilers to implement dynamic binding since it introduces a small and fixed overhead. One dispatch table is associated with each class. A dispatch table contains the run-time signatures of the methods of the class. For efficiency reasons the method signature is generally the address of the method. Dispatch tables are also known in the C++ literature as *Vtables* (for Virtual methods Table). The layout of the GNAT dispatch tables is described in various papers such as [M06] and [M07].

Each object instance has a hidden component (the Vtable pointer in C++ and Java, or the Tag in Ada) that references the dispatch table corresponding to the actual class of the object. At the point of a dispatching call, the compiler generates code that uses this hidden component to

1. get the dispatch table associated with the object,
2. index it by a number associated with the method signature (a constant known at compile time), and
3. make an indirect invocation of the target method.

With such an implementation, dispatching calls are deterministic and bounded in time, with a performance similar to an indirect call.

Dynamic dispatching has potential safety and security problems since the wrong initialization of these tables or the corruption of their contents can lead to improper control flow of dispatching calls.

In order to prevent corruption of the dispatch table content during the execution of the code, GNAT generates dispatch tables of library level classes as static data that is placed in a read-only data section of the object code. Appropriate linker switches can be used to ensure that such sections are placed in ROM. Placing such tables in ROM insures that they cannot be modified either inadvertently or maliciously.

DIRECTIONS FOR GNAT PRO USERS

As shown in the previous chapters, there are many issues to consider when planning the use of Object Oriented features in a High Integrity context, but Ada's design offers safe solutions that have made Ada a language of choice in this area. Indeed, GNAT Pro High-Integrity Edition customers have already used Ada to develop Object Oriented software that has passed the highest level of avionics certification.

The following are main points to consider when starting a new project using OO technology. Choices need to be made based on the desired generality of the OO features and the required level of certification or safety case.

- *Deciding degree of formality.* What amount of rigor is desired or required for the verification activities? If formal proofs of properties are of interest, the *SPARK Pro* technology should be considered. See *Using SPARK Pro for Formal Proofs*.
- *Choosing appropriate run-time library.* In order to reach a high level of confidence, one of the High Integrity run-time libraries should be adopted. Usually this choice is constrained by considerations more global than those related to OO features, such as the necessity of having specific certification evidence for the given library. Nonetheless, one still needs to verify that the chosen runtime provides support for OO features compatible with the project needs and expectations. See *OO Support in GNAT Pro High-Integrity Profiles*.
- *Mitigating OO vulnerabilities.* The project needs to define a strategy that addresses the complexities and vulnerabilities defined earlier, and to select the activities and tools that will help implement this strategy. Particular care must be given to solving the vulnerability concerning dynamic dispatch described in the *Dynamic Dispatch and Substitutability* section. This section already contains references to elements of the GNAT Pro technology that can be used for this purpose.
- *Defining a coding standard.* One key element that will help simplify the implementation of the strategy mentioned above is to define and enforce a Design and Coding Standard, with the goal of limiting the complexity of the application and thus reducing the difficulty in achieving safety verifications. The section *Writing a Coding Standard with GNATcheck* addresses this issue.

7.1 Writing a Coding Standard with GNATcheck

GNAT Pro offers a mechanism for automating the verification of a coding standard through the use of the GNATcheck tool. The coding standard is materialized by a file containing GNATcheck rules. Here is a simple example of such a standard:

```
-- This is a simple Coding Standard
+RGOTO_Statements
+RMetrics_Cyclomatic_Complexity : 5
+RRestrictions : No_Dependence => Ada.Containers
+Style_Checks:0
```

We can see that a coding Standard file of this sort can include comments like an Ada source. Non comment lines include one rule and optionally its parameters. The leading string is either '+R' or '-R' in order to enable the rule or disable it. The conventions for GNATcheck rules is that they describe the constructs or patterns that are to be flagged. For instance the first rule "GOTO_Statements" requests GNATcheck to emit a message for each GOTO statement in the Ada sources that are analyzed. The second rule "Metrics_Cyclomatic_Complexity : 5"

is a good example of a parametrized rule: it will requests GNATcheck to emit a message for each subprogram whose cyclomatic complexity is greater or equal to 5. The last 2 rules in this example show how GNATcheck can take advantage of GNAT's comprehensive list of Restrictions and style checks. The first one would have an effect similar to adding to the Ada sources:

```
pragma Restrictions (No_Dependence, Ada.Containers);
```

The second one would have an effect similar to compiling the Ada Sources with option `-gnatyO`. The advantage of using the GNATcheck version of those rules as opposed to the equivalent features of the compiler is that all the generated messages will be collected in the GNATcheck report that can be used as evidence of the level of adherence to the coding standard. GNATCheck is indeed a qualifiable tool that is regularly used for fulfilling certification obligations. Note that GNATcheck also provides a mechanism to deal with accepted deviations from the standard called *exemptions*. More information can be found in the GNATcheck User's Guide [GNATcheckUG] concerning GNATcheck usage as well as all the available rules. The next section will focus on the rules that concern directly one of the issue addressed in this paper.

7.1.1 Relevant GNATcheck Rules

- `Controlled_Type_Declarations`

This rule prevent the use of Controlled types. Note that such types are not supported by GNAT Pro HIE runtimes so when the profile associated with such runtimes is already part of the Coding Standard such a rule might be considered redundant.

- `Deep_Inheritance_Hierarchies`

This rule has an argument: the longest acceptable derivation path. It allows controlling the complexity of the class hierarchy (depth) and it can be used both in single and multiple inheritance contexts.

- `Too_Many_Parents`

This rule has an argument: the maximum number of parents. It is used to control the complexity added by the use of multiple inheritance. It can also be used to ban multiple inheritance altogether when the parameter is set to one.

- `Direct_Calls_To_Primitives`

This rule can be used to prevent static dispatch in order to address the vulnerability describe in the *Static Binding and Overriding* section. It solves the vulnerability by verifying that the *simple dispatch rule* is applied.

- `Visible_Components`

This rule prevent the use of composite types whose component type is publically known. So it encourages the use of private types and help addressing the vulnerability described in the *Lack of Encapsulation* section.

- `Style_Checks:0`

This rule detects all overriding primitives that are not declared with the `overriding` keyword and thus enforces the suggested strategy for addressing the vulnerability described in the *Improper Overriding* section.

- `Restriction:No_Dispatch` and `Restriction:No_Dispatching_Calls`

The first restriction is part of the Ada standard. It prevents all uses of class-wide types, which effectively prevents the use of polymorphism but also prevents any attempt as classwide programming even when it doesn't trigger dynamic dispatch. The second one, which is specific to the GNAT Pro technology is less restrictive, and allows the use of class-wide types and operations, but disallows any form of dispatching calls.

7.2 Using SPARK Pro for Formal Proofs

SPARK [O11] is a high-level, high-integrity software development language, supported by powerful tools. The compilable elements of SPARK are a subset of Ada, but SPARK is not just a subset of Ada: an integral part of the language is its annotation language, which forms an essential part of the “contractual” specification of SPARK programs.

7.2.1 Language and Tools

Part of a program’s contract, in both Ada and SPARK is the set of *signatures* for each of its (visible) subprograms: its name, formal parameters (together with their types and modes) and description (in accompanying comments). SPARK programs have additional contractual elements, however. These additional elements consist of a set of core annotations, which may also be supplemented by more specific proof annotations. The core annotations allow a fast, polynomial-time analysis of SPARK source code to check for data-flow and information-flow errors, which can indicate a failure of the code to meet its contract. The proof annotations are optional; when these are used, they can support the mathematical proof of properties of the source code. The proofs performed can range from proof of exception freedom, at its simplest, all the way to a proof of correctness against a formal specification.

The subset of Ada which is at the heart of the SPARK language has been chosen to produce a simple yet powerful programming language, retaining the key features that support the construction of software that is demonstrably correct and abstraction through specification. For example, packages, private types and separate compilation are all important aspects of the language. The verification of the body of each package is independent of the bodies of any other packages: only the contracts of the packages on which it depends are needed for verification purposes. In particular this allows the programmer to use package bodies developed in SPARK together with package bodies developed in full Ada, provided the package specifications are in SPARK.

7.2.2 Formal Proofs

Returning to the stack example, the specification of `Push` seen previously can be written as follows in SPARK, with proof annotations in special comments introduced by a `#` symbol.

```
package IStacks is

  subtype Item is Integer;
  Max : constant := 100;

  type Stack is private;

  function Is_Empty (S : Stack) return Boolean;
  function Is_Full  (S : Stack) return Boolean;

  function Size (S : Stack) return Natural;
  --# return Result => (Is_Empty (S) <-> Result = 0)
  --#                    and (Is_Full (S) <-> Result = Max)
  --#                    and Result in 0 .. Max;

  procedure Push (S : in out Stack; I : in Item);
  --# pre  not Is_Full (S);
  --# post Size (S) = Size (S~) + 1;

private
  ...
end IStacks;
```

Note the special operators in SPARK annotations, such as the equivalence `<->` and the reference to the value of variable `S` at subprogram entry, denoted `S~`, similar to `S'Old` in Ada 2012.

Given a suitable implementation of `Push` in SPARK, the SPARK tools generate logic formulas that should be true for function `Push` to respect its contract. Then, various provers provided in the SPARK toolset can be used to

automatically prove that these logical formulas are true.

Of special interest in object-oriented programming is the verification of the Liskov Substitutability Principle. For every overriding subprogram, SPARK generates formulas that ensure that 1) the overriding subprogram precondition is implied by the overridden subprogram precondition, and 2) the overriding subprogram postcondition implies the overridden subprogram postcondition.

7.2.3 Control and Data Coupling

As discussed previously, verifying that preconditions and postconditions are respected is an effective means of controlling software components' control coupling. Formal verification, in contrast to testing, gives the additional guarantees that all execution paths have been covered in this verification.

SPARK also provides a straightforward solution to checking data coupling, through the static verification of data-flow and information-flow. The user states in annotations the expected reads and writes of subprograms, or the flows between reads and writes, and the SPARK toolset checks these automatically. This guarantees that no use of data can occur silently (i.e., without being explicitly specified).

Data flow analysis concerns how data flows within and between programs. If we always, or sometimes, use the value of a variable before it has been assigned a value, for instance, this is a data flow error. Data flow analysis is performed by the SPARK Examiner tool on all of the source code to which it is applied.

Information flow analysis concerns how information flows between variables. For a given program, we can determine a set of inputs (parameters of mode **in** or **in out**, together with global variables from which we may read values) and a set of outputs (parameters of mode **in out** or **out**, together with global variables that we may modify). Information flow analysis considers which inputs may affect which outputs, both for individual subprograms and, by composition, subprograms which call these, and so on. Information flow analysis can also detect more subtle programming errors, including missing or unexpected dependencies between inputs and outputs, and loop exit conditions which may become 'stable' (and cease to change) after a small number of iterations. Information flow analysis is optional, but can be applied to all SPARK code that has the relevant core annotations.

As an example, here is a SPARK specification of a procedure `Exchange` which swaps the values of `X` and `Y` and stores `X` in some global `G`:

```
procedure Exchange (X, Y : in out Float);
--# global out G;
--# derives X from Y &
--#         Y from X &
--#         G from X;
```

7.3 Using GNATstack for Stack Resource Analysis

High-Integrity systems must provide evidence demonstrating the impossibility of stack overflow, and the use of object-oriented capabilities (namely the dynamic binding) complicates this calculation of stack usage.

The stack is the memory area that stores local information for the subprograms being executed. A stack overflow occurs when there is an attempt to store more data on the stack than what can fit. The consequences of a stack overflow depend on the operating environment and execution context. If this event is properly detected and handled, execution can proceed in error-recovery mode. If this situation remains unnoticed, memory can be corrupted, leading to unpredictable execution. Hence, it is of paramount importance to avoid stack overflows in the first place by means of analyzing worst-case stack requirements.

The two approaches to stack requirement analysis are based on either dynamic testing or static analysis techniques. Dynamic testing-based approaches usually involve measuring the maximum amount of memory used while running or simulating the application. Static analysis techniques entail computing per-subprogram stack consumption combined with control-flow analysis.

The major weakness of dynamic testing-based approaches is that they cannot guarantee that the worst-case execution path has been exercised during the testing campaign. Static stack analysis techniques provide worst-case

results that can be safely used for dimensioning stacks, ensuring reliable stack memory usage and thus guaranteeing safe execution.

Static stack analysis techniques need to be able to know all feasible control flow during execution, and this is where the dynamic binding mechanism creates problems: the target of a dispatching call depends on the actual context of execution, and it cannot, in general, be known statically.

The approach to handle statically dispatching calls is to identify the set of overriding primitive operations (methods) that can potentially be called. Taking into account the class hierarchies and associated primitive dispatching operations, and the class-wide type of the controlling operand, the stack analysis tool can determine the list of potential primitive operations that can be the targets of any dispatching call.

For example, with the *Shape* class hierarchy, if we have the following code:

```

procedure Get_Name (Object : Shape) is
  Name : String (1 .. 5);
begin
  Name := "Shape";
  ...
end Get_Name;

procedure Get_Name (Object : Polygon) is
  Name : String (1 .. 7);
begin
  Name := "Polygon";
  ...
end Get_Name;

procedure Get_Name (Object : Rectangle) is
  Name : String (1 .. 9);
begin
  Name := "Rectangle";
  ...
end Get_Name;

procedure Use_Name (Object : Shape'Class) is
begin
  Get_Name (Object);
  ...
end Use_Name;

```

Procedure `Use_Name` performs a dispatching call which can only be resolved at execution time to any of the three primitive operations associated to the `Shape'Class` type hierarchy. The way to address this issue statically is to consider the three primitive operations as potential targets of the dispatching call. Therefore, the procedure:

```

procedure Use_Name (Object : Shape'Class) is
begin
  Get_Name (Object);
  ...
end Use_Name;

```

is conceptually considered equivalent, from the point of view of static control-flow analysis, to the following:

```

procedure Use_Name (Object : Shape'Class) is
begin
  if Object'Tag = Shape'Tag then
    Shape (Object).Get_Name;
  elsif Object'Tag = Polygon'Tag then
    Polygon (Object).Get_Name;
  elsif Object'Tag = Rectangle'Tag then
    Rectangle (Object).Get_Name;
  end if;
  ...
end Use_Name;

```

When using *GNATstack* [GNATSTACK], AdaCore's static stack analysis tool, dispatching calls are analyzed as described above, obtaining a safe upper bound for the worst-case path in terms of stack usage. For the previous example, if we want to focus on the path starting from `Use_Name`, we get the following stack usage information:

Accumulated stack usage information for entry points

```
shapes.use_name : total 208 bytes
+> shapes.use_name at Use_Name:shapes.adb:31:4 : 96 bytes
+> dispatching to shapes.get_name at Get_Name:shapes.adb:23:4 : 112 bytes
```

List of dispatching calls resolved by analysis

From subprogram `shapes.use_name` at `Use_Name:shapes.adb:31:4`:

```
Call at shapes.adb:33:7 may dispatch to:
+> shapes.get_name at Get_Name:shapes.adb:7:4
+> shapes.get_name at Get_Name:shapes.adb:15:4
+> shapes.get_name at Get_Name:shapes.adb:23:4
```

In this case, the worst-case path is the one where the dispatching call is resolved to the primitive operation associated with the tagged type `Rectangle` (defined at `shapes.adb:23:4`). We can see in the previous example that this primitive operation is declaring the largest object on the stack.

GNATstack also provides the list of possible targets for every dispatching call. This list can be exploited to determine whether the implementation corresponds to the model, and whether the testing campaign has covered all possible targets for dispatching calls.

The same principles described here apply also to multiple inheritance of interfaces, with *GNATstack* able to statically analyze this kind of construct.

Therefore, *GNATstack* handles dynamic dispatching by considering all possible targets of dispatching calls (according to the hierarchies of classes and primitive dispatching operations, and the class-wide type of the controlling operand). With this information, without doing any data-flow analysis based on the context of the execution, it constructs statically the application call graph, so it can explore all possible paths searching for worst-case stack usage.

GNATstack takes into account Ada tasks when performing the control-flow analysis, presenting per-task stack requirements which can be used to allocate the stack space required by each task in the application (using `pragma Storage_Size (Max_Size)` in the task declaration.)

7.4 OO Support in GNAT Pro High-Integrity Profiles

The GNAT Pro High-Integrity Edition defines four predefined profiles:

1. The *Zero Footprint Profile*, an Ada subset requiring no run-time support;
2. The *Cert Profile*, comprising the features in the Zero Footprint profile together with a restricted set of thread-safe features, in particular exception propagation;
3. The *Ravenscar Profiles*, comprising the features in the Zero Footprint profile (Ravenscar ZFP) or the Cert profile (Ravenscar Cert) together with a restricted set of tasking features; and
4. The *Full-Runtime Profile*, comprising the complete Ada language.

The Zero Footprint Profile, the Cert Profile and the Ravenscar Profiles are collectively known as the *High-Integrity Profiles*, since they are designed to be used in applications that need to be certified for safety-critical use. The following Ada features for Object-Oriented Programming are supported under the various High-Integrity profiles:

- Tagged types defined at library level are supported; tagged types defined in nested scopes are not supported since their implementation require compiler support to handle pointers to primitives declared in nested scopes, and runtime support to elaborate nested tagged types in presence of multiple tasks (See Sections *Minimize Need for Ada Run-Time Support and Facilitate Source-to-Object Traceability* and *Robustness of*

Dynamic Dispatch Mechanism.) In addition, library-level tagged types are statically elaborable and thus can be used in the presence of the restriction `No_Elaboration_Code`.

- Tagged objects can be declared at library level, in nested scopes, or allocated in the heap. In the ZFP profile, allocation of tagged objects in the heap is under control of the programmer, who needs to provide the low-level memory management routines invoked by the code generated by the compiler to allocate and free the memory of the objects. The Cert and Ravenscar profiles provide a limited version of function `gnat_malloc` that simply calls the underlying `malloc` routine. Allocation and deallocation of objects may be limited through the following restriction identifiers defined in the Ada Reference Manual: `No_Local_Allocator`, `No_Allocators`, `No_Implicit_Heap_Allocations`, and `No_Unchecked_Deallocation`.
- Declarations of class-wide objects (i.e., polymorphic objects) and class-wide parameters are supported. They can be limited through the Ada restriction identifier `No_Dispatch` if the safe use of record extensions is the unique object-oriented requirement of the application.
- Dynamic dispatching is supported and can be limited through the GNAT restriction identifier `No_Dispatching_Calls`, which also forbids calls internally generated by the compiler (for example, implicit dispatching calls generated by the compiler to handle class-wide object assignment or class-wide object comparison).
- The object-operation notation is fully supported under any profile since its use does not involve extra cost in the code generated by the compiler.
- A subset of interface types is supported in the High-Integrity profiles. Various restrictions on interfaces help reduce the run-time support:
 1. No task interfaces, protected interfaces or synchronized interfaces,
 2. No dynamic membership test applied to interfaces (only those cases in which the evaluation can be performed at compile-time are supported),
 3. No class-wide interface conversions,
 4. No declaration of tagged type covering interfaces in which its parent type has variable-size components, and
 5. `'Address` not supported on objects whose visible type is a class-wide interface.
- Streams are partially supported. That is, dispatching calls associated with streams are not allowed to avoid the need for run-time support to handle dispatching stream operations. However, use of the package `Ada.Streams` (or another package that does so itself) in the High-Integrity application is permitted as long as no actual stream objects are created and no stream attributes are used.
- Controlled types are not supported since they require extensive run-time support.
- A limited version of package `Ada.Tags` is available in the default implementation of these profiles. Services `Wide_Expanded_Name`, `Internal_Tag`, `Descendant_Tag`, `Is_Descendant_At_Same_Level` and `Interface_Anccestor_Tags` are excluded since their implementation involves extra run-time cost.
- Generic dispatching constructors are not supported since their implementation requires the above-mentioned routines.

CONCLUSION

In this document, we have presented the general Object Oriented concepts and their incarnation in Ada including its most recent revisions. Ada is a language that is generally considered as well suited for the development of High Integrity Systems. So this first level of description already tackles some of the safety aspects related to the style of programming associated with OO in the context of the relevant language construct. We have, then, reviewed the most common safety-related vulnerabilities associated with OO and proposed ways of addressing or at least mitigating them. Following chapters discuss other safety-related issues such as complexity management or how to simplify specific analyses that may be requested when certification is required. In particular, we have also emphasized the new Ada support for contract programming and how it can be used for some of such analyses. Finally we have provided specific directions to GNAT pro users for activities or analyses based on GNAT Pro & SPARK Pro tools.

The goal of this document is to gather all available information related to the safe use of OO technology with Ada, and more particularly in the context of the GNAT Pro technology. We expect this document to evolve over time, so we kindly ask readers to provide as much feedback as possible to AdaCore at report@adacore.com with the name of the document mentioned on the subject line.

BIBLIOGRAPHY

- [Ada2005] *Ada 2005 Language Reference Manual*. <http://www.adaic.org/ada-resources/standards/ada05/>
- [C06] C. Comar, R. Dewar, G. Dismukes. “Certification & Object Orientation: The New Ada Answer”, *ERTS 2006*. http://www.adacore.com/wp-content/uploads/2006/03/Certification_OO_Ada_Answer.pdf
- [GNATcheckUG] *GNATcheck User’s Guide*. <http://www.adacore.com/category/developers-center/reference-library/documentation/>
- [GNATHIE] *GNAT Pro User’s Guide Supplement for High-Integrity Edition Platforms*. http://www.adacore.com/wp-content/files/auto_update/gnat-hie-docs/html/gnathie_ug.html
- [GNATSTACK] J.F. Ruiz, E. Botcazou, O. Hainque, and C. Comar. “Preventing Stack Overflow using Static Analysis”, *Proceedings of DASIA 2007, Data Systems In Aerospace*, Naples, Italy, 2007.
- [LW94] 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.
- [M06] J. Miranda. *The Implementation of Ada 2005 Interface Types in the GNAT Compiler*, http://www.adacore.com/wp-content/uploads/2006/06/Ada-2005_Interface_Types.pdf
- [M07] J. Miranda. “Towards Certification of Object-Oriented Code with the GNAT Compiler”, *Ada User Journal*, Volume 28, Number 3, September 2007. <http://www.adacore.com/2011/01/05/towards-certification-of-object-oriented-code-with-the-gnat-compiler/>
- [M97] B. Meyer. “Object-Oriented Software Construction”, *Prentice Hall Professional Technical Reference*, 2nd Edition, 1997, pp 331-410.
- [O11] I. O’Neill. *SPARK - a language and tool-set for high-integrity software development*, To appear, 2011.
- [OOTiA] US Federal Aviation Administration. *Handbook for Object-Oriented Technology in Aviation*, October 2004. http://www.faa.gov/aircraft/air_cert/design_approvals/air_software/oot
- [RE2010] RTCA WG 205 / EUROCAE SC 71. Information Paper Number 61, Revision E, *Object-Oriented and Related Technologies Supplement*, April 2010.