Certification & Object Orientation: The New Ada Answer

Cyrille Comar[1], Robert Dewar[2], Gary Dismukes[3]

1: comar@adacore.com, AdaCore, 8, rue de Milan, 75009, Paris, FRANCE
2 : dewar@adacore.com, AdaCore, 104 Fifth Ave., New York, NY 10011, USA
3 : dismukes@adacore.com, AdaCore, 104 Fifth Ave., New York, NY 10011, USA

Abstract:

The object model of Ada 2005 is well-suited for applications that have to meet certification at various levels. We review the use of Ada in the context of certification, and show that the object-oriented facilities of the current language standard, properly restricted to avoid dynamic dispatching, can already be used without problems under current DO-178B guidelines. We then examine the complications to certification that are presented by dynamic dispatching in a single inheritance model, and show implementation-specific ways of addressing these complications. Finally, we discuss the problems introduced by the use of multiple inheritance. We conclude by showing how, regardless of the extent to which object-oriented idioms are used, Ada provides a safe and efficient vehicle to create certifiable systems.

Keywords: Ada, DO-178B, Certification, Object-Oriented Programming.

1. Introduction

Software construction has evolved considerably in the last two decades. One of the most notable advances is the use of object-oriented techniques in commercial applications. The enhancement of reusability, adaptability and maintenance has led to wide deployment of these techniques in many application domains.

Languages such as C++, Java, and C# have evolved over the last decade and, thanks to them, object-oriented programming has become a widely used paradigm. Over the same period Ada has evolved to incorporate object-oriented features into its original type model. The Ada 95 standard added to Ada 83 tagged types, single inheritance, polymorphism, and dynamic dispatching. The latest revision of the language, known as Ada 2005, adds multiple inheritance of interfaces and numerous other object-oriented programming idioms.

There is currently a trend towards the use of object-oriented techniques in the construction of high-integrity software systems, such as in avionics airborne systems. In this particular domain, the integration of best practices in software construction must also be balanced with the need to increase confidence in the safety of the system. A further feature to note is that Avionics software has a significantly longer life-cycle than software in other application areas.

The aim of the RTCA DO-178B [5] standard is to provide guidelines for building airborne systems that perform their intended functions with the appropriate level of safety. Developed in the 1980s and finalized in the early 90s, DO-178B is based on the waterfall software process that was considered best-practice at the time the standardization process in this field began. One of the objectives of the forthcoming revision of this standard, DO-178C [7] (still in the early stages of development), is to take into account object-oriented techniques and their associated development processes. A preliminary document, the Handbook for Object Oriented Technology in Aviation (OOTiA) [1], already provides a comprehensive analysis and specific guidelines on how to address the safety concerns associated with object-oriented techniques in the context of DO-178B [5].

This paper reviews the most relevant features of the Ada language for certification and summarizes the evolution of object-oriented features in Ada. it then shows how Ada is ideally suited for use in certified applications that use object-oriented features, in particular when complemented with certification-oriented facilities of the development environment.

2. Ada and Certification

Since the emergence of the RTCA DO-178B standard [5], Ada has been one of the few languages of choice for the construction of airborne systems thanks to its clear semantic definition and strong typing model. It has been used successfully in many major aeronautics projects (Boeing 777, A340, and more recently Boeing 787, A380 and A400M). Clear semantic definition is considered a desirable precondition for minimizing non-determinism by DO-248B [6] FAQ#32 *(What are Defensive practices*), where one can read:

Avoidance of non-determinism: Non-determinism may be reduced through one or more of the following practices:

1. *Choose a language with a well-defined standard.*

Not many languages besides Ada can claim a well-defined official standard. Fewer can claim that all major implementations fully adhere to the standard. And fewer still have an official and standardized test suite [19] supporting validation of this claim. It is also notable that Ada has several guides for its use in high-integrity systems [10], [17].

Ada's clear semantic definition has also fostered the wide use of ASIS [16], which considerably eases the creation of semantics-oriented verification tools. Such tools can be qualified in order to produce evidence that will help meet the DO-178B objective 6.3.4d:

d. Conformance to standards: The objective is to ensure that the Software Code Standards were followed during the development of the code, especially complexity restrictions and code constraints that would be consistent with the system safety objectives.

For example, consider a tool that is part of the standard GNAT toolset: `gnatmetric` computes industry-standard source metrics such as McCabe cyclomatic and essential complexity [13]. Another example is `gnatcheck`, an ASIS-based tool, whose purpose is to produce certification evidence that specific semantic rules are followed; or when they are not followed, that violations are clearly identified so they can be justified. This tool is designed to be easily extensible for specific project standards. It already implements dozens of local rules, such as "*no goto statements*" as well as several more global ones such as "*no side effects in functions*" (i.e., no direct or indirect modification of a variable whose scope is wider than the function body) and "*no recursion*" (i.e., no cycles in the subprogram call graph).

Ada's pragma *Restrictions* ([2] 13.12) is also a significant aid for enforcing the "*Conformance to standards"* objective mentioned above. For instance, dynamic memory allocation, whether implicit or explicit, can be excluded from the application by simply specifying:

```
pragma Restrictions (No_Allocators);
pragma Restrictions
                (No_Implicit_Heap_Allocation);
```

Furthermore, the compiler can take advantage of pragma Restrictions to produce simpler and more efficient code. For instance, in the presence of

```
pragma Restrictions (No_Abort_Statements);
pragma Restrictions
   (Max_Asynchronous_Select_Nesting => 0);
```

A good implementation will not generate the abort-deferral code that is otherwise mandated by the language. Not only can performance improvements be expected, but more importantly, deactivated compiler-generated code can be reduced, thus easing the effort of code verification and justification for high levels of certification.

There are several interesting subsets of Ada for high-integrity systems. One of them is the Ravenscar profile [18], a subset of the tasking features of Ada which is powerful enough for real-time programming but simple enough to be amenable to static analysis for certification, and that can be supported by a small, reliable run-time system. Another notable example is SPARK [11] that includes Ada constructs regarded as essential for the construction of complex software, but removes all the features that may jeopardize the requirements of verifiability, bounded space and time, and minimal run-time system.

Thanks to Ada's strong typing and subtyping model, run-time checks are generated that can help detect malfunctions earlier, during the testing phase. At high levels of certification, some projects choose not to insert run-time checks in the executable to be certified. This choice can be justified if static analysis tools such as SPARK [11] have been used to provide proof that specific run-time errors cannot occur. Check elimination can also be motivated by the desire to minimize deactivated compiler-generated code, or by the difficulty of implementing a reliable recovery mechanism, as shown by the Ariane 5 launch disaster analysis [14]:

Although the source of the operand error has been identified, this in itself did not cause the mission to fail. The specification of the exception-handling mechanism also contributed to the failure.

Even when compiler checks are not generated in the final executable, they can play a beneficial role, together with **`pragma`** `Normalize_Scalars` ([1] H.1), for satisfying objective 6.3.4f of DO-178B [5] (use of uninitialized variables or constants). **`Pragma`** `Normalize_Scalars` instructs the compiler to initialize scalar variables that do not have explicit initialization with a deterministic value (outside its range when possible), in order to increase the likelihood of generating a repeatable run-time fault when the variable is read before receiving a value. Specific compilers can provide more extensive and comprehensive run-time checks to maximize this possibility. [9] describes such an implementation and

its successful use in an industrial non-safety-critical context (Air Traffic Flow Management). Running a full requirements-based and structural-based testing campaign in such a mode is likely to significantly raise confidence in attaining objective 6.4.3f. For the results of such a process to be acceptable as certification evidence, this compiler functionality does not require full certification since it is not involved in the production of the final code. It only needs to be qualified according to DO-178B section 12.2.2 [5].

Finally, it is worth mentioning that Ada's strong typing system is also known to offer useful capabilities for tools such as static stack-usage analyzers. In particular, [8] mentions how appropriate type information can be used to compute the maximum stack allocation requirements for variable size objects. It can also help computing complete call graphs even in the presence of indirect calls.

3. Brief Review of the Ada Object Model

Ada is a general-purpose language that has added support over time for a range of programming paradigms: procedural programming, functional programming, object-based programming, and object-oriented programming with both single inheritance (type extension and dispatching operations in Ada 95) and multiple inheritance (inheritance of interfaces in Ada 2005).

In Ada 83, the object model was limited to data encapsulation and information hiding, using private types and packages to separate specification from implementation. For example, the `DisplayElement` class of [1] (pp. 3-77) could be represented as:

```
package P is
   type Element is private;
   procedure Draw      (X : Element);
   procedure Highlight (X : Element);
   procedure Hide      (X : Element);
private
   type Element is record
      …
   end record;
end P;
```

Here the client interface to the `Element` abstraction is given in the visible part of the package by a private type and a set of operations. The details of the implementation are given separately in the private part and body of the package, effectively insulating the user of the abstraction from the representation and implementation of the type and its operations.

In Ada 95, the language was revised by adding full-fledged object-oriented support in the form of single inheritance, with type extension and dynamic dispatching. This was done in an upward-compatible way, building on Ada 83's existing framework for package encapsulation and derived types. By declaring a type to be tagged and defining a set of primitive operations for the type, the type can subsequently be extended. Such a type extension inherits the data components and primitive operations of the parent type, and can extend the parent type by adding components and operations, as well as by overriding any inherited operations. The notion of type classes was also introduced, to provide polymorphism of objects and operations. Each tagged type `T` thus serves as the representative root type of a potential hierarchy of the types that extend it, where this set of types is designated `T'Class`.

The `Element` abstraction could be represented by the following tagged type:

```
package P is
   type Element is tagged private;
   subtype Any_Element is Element'Class;
   procedure Draw (X : Element);
   …
end P;
```

The keyword **`tagged`** in the declaration of type `Element` identifies the type as being extensible. Element is the root type of a class of types that could be derived from `it`, while `Element'Class` denotes the entire set of types in that class. The type `Element` can now be extended, for example by defining a specialization:

```
package Pictures is
   type Picture_Element is
      new P.Element with private;

   procedure Draw (X : Picture_Element);
   procedure Brighten
     (X : Picture_Element; Y : Intensity);
   ...
private
   type Picture_Element is new P.Element
      with record
         Brightness : Intensity;
      end record;
end Pictures;
```

The type `Picture_Element` is an extension of Element. In this case the new type is declared as a private extension whose full type adds a single new component (hidden from clients of the type). The operation `Draw`, inherited from the parent type, is overridden by an implementation specialized for the new type (though that implementation might also invoke the parent type's operation). Other operations of the parent type can be inherited as is or overridden, and additional primitive operations can be declared for the type extension (such as the procedure `Brighten` in this example).

Subprograms declared together with a tagged type in the same package and having at least one parameter (or result) of the tagged type are called dispatching operations of the type. A call to such an operation is not necessarily dispatching however. The call will only dispatch when invoked with an actual parameter whose type is the class-wide type of the associated type class.

As an example of dynamic dispatching, consider the following class-wide subprogram (called class-wide because its parameter will accept arguments of any type in the class, such as `Element`, `Picture_Element`, etc.):

```
procedure Draw_Any (X : Any_Element) is
begin
   Draw (X);            -- dispatching call
   Draw (Element (X));  -- normal call
end Draw_Any;
declare
   Elt   : Element;
   Pixel : Picture_Element;
begin
   Draw_Any (Elt);
   Draw_Any (Pixel);
end;
```

The first call to `Draw` from within `Draw_Any` will dispatch to the appropriate implementation of `Draw` associated with the underlying value of the parameter `X`, which might be of any type in the class of types rooted at `Element`. The second call involves a conversion of the class-wide parameter `X` to the specific root type `Element`, and so will directly invoke the procedure `Draw` associated with type `Element` (even though the underlying value might be of some other type in the class).

As an extension to the basic capability for type classes, Ada 95 also added a facility for declaring so-called "controlled types". A controlled type allows overriding of three special operations for initialization, finalization, and adjustment (as part of an assignment) of objects. This feature provides the designer of an abstraction with additional control and flexibility in managing objects of the type by ensuring that any resources associated with an object are properly initialized upon creation and cleaned up when the object is no longer needed.

The most recent revision of the language, known as Ada 2005, introduces several new capabilities that enhance the existing object-oriented features. These include:

- the ability to conveniently define mutually dependent types in separately compiled packages (limited with clauses)
- generalized uses of Ada 95's anonymous access types to reduce the need for explicit conversions between access types
- allowing type extensions to occur at deeper nesting levels than the parent type, extending their usefulness without compromising safety
- the ability to differentiate new operations explicitly from overriding ones, as in this example:

```
with P;
package Q is
   type Textual_Element is
      new Element with private;

   overriding procedure
     Draw (X : Textual_ Element);
   not overriding function
     Get_Displayed_Value (X : Textual_Element)
       return String;
   ...
end P;
```

`Textual_Element` derives from `Element` and thus inherits its primitive operations. It can override some of them, such as `Draw`, as well as define new ones, such as `Get_Displayed_Value`. The keyword **`overriding`** makes the choice clear to the reader and allows the compiler to detect inadvertent typos that could otherwise be interpreted as overloaded operations.

Ada 2005 also introduces the conventional object prefix notation for invoking operations on objects, permitting programmers to use `object.method (param);` as an alternative to the Ada 95 functional form `method (object, param);`, as illustrated by the following calls:

```
X.Draw;
Put_Line (X.Get_Displayed_Value);
```

Finally, Ada 2005 introduces a Java-like limited form of multiple inheritance through the keyword **`interface`**. The interface feature enables inheritance of specifications (roles) rather than implementations, avoiding the complexity and danger of the more general forms of multiple inheritance.

An interface as defined in Ada 2005 is very similar to an Ada 95 abstract tagged type which has no components and each of whose operations is either abstract or null. The type extension rules are enhanced to allow tagged types (as well as interfaces) to inherit from any number of interfaces. In order to preserve single inheritance of implementations, a tagged type extension must still derive from a single tagged type ancestor. When a tagged type inherits from one or more interfaces, it must override any abstract operations inherited from

those interfaces (unless the tagged type itself is abstract).

4. Simple Object Model (without Dispatching)

In their current form, certification standards such as DO-178B [5] are wary of dynamic dispatching. If its use is not formally banned, it is strongly discouraged by the afore-mentioned FAQ 34 in DO-248B [6] which explicitly states:

> *Avoidance of non-determinism: Non-determinism may be reduced through one or more of the following practices:*
>
> *…*
>
> *4. Avoid use of dynamic binding…*

Dynamic binding is a synonym for dispatching (see [1] 1.3.5). Non-determinism introduced by dispatching will be discussed in the next section along with issues related to proper testing of such constructs. Although solutions to these issues are emerging, they are not yet fully established. The same wariness caused dispatching to be excluded from the first version of the Ada standard. Now that a complete object-oriented paradigm is supported in Ada, the question arises: can it be of any use to those who prefer not to have to deal with the certification difficulties related to dispatching? Ada's answer is clearly "yes". Type extension and inheritance are powerful mechanisms for defining closely related abstractions and can be used without dispatching.

The "`object.method`" notation is also considered a significant syntactic improvement in some situations, particularly when it simplifies the source code and thus enhances its readability. This is especially true where local coding standards prohibit use clauses. This is the case in SPARK [11], which requires that any entity always be referenced using exactly the same syntactic form thus entailing the use of fully qualified names. When a method defined in a different package is called on a local object, the difference of notation can result in substantial simplification:

```
Parent.Child.Grandchild.Method (Obj, Param);
```

compared to the much simpler:

```
Obj.Method (Param);
```

In Ada 2005, this prefixed notation is only available for primitive or class-wide operations of tagged types, which encourages their use even when dispatching is prohibited. Such a prohibition can easily be enforced by using the standard restriction:

```
Pragma Restrictions (No_Dispatch);
```

This is a configuration pragma applying to all units in a partition. Such a restriction can also be used by certification-friendly implementations to avoid generating all the implicit code required to support dispatching properly. For instance, the so-called "tag" component no longer needs to be materialized in tagged records. Dispatch tables do not need to be generated, nor filled during type elaboration. Implicit dispatching operations such as `"="`, `'Size`, `'Read`, and `'Write` can be avoided. These operations usually need to be created unconditionally for each tagged type, in order to support potential dispatching calls from different units. Such simplification of the generated code reduces the need for localizing and potentially needing to justify deactivated compiler-generated code as already alluded to in section 2.

5. Towards Certification of Dispatching Calls

Section 3 showed that the programmer can choose whether a given call to a primitive operation will dispatch. This flexibility can be used to limit the number of dispatching calls, thereby limiting their associated certification cost. This flexibility is not available in Java, where all operation invocations are dispatching (unless a routine is declared as final – similar to an Ada class-wide operation). It is available in C++, but at the cost of forcing the programmer to indicate whether an operation itself (not a specific call) is virtual. A virtual operation will potentially always dispatch while a non-virtual one will never dispatch. C++ implementations are allowed to optimize dispatching calls into regular calls when the context permits, but this is not under the control of the developer.

It is also worth noting that most OOTiA [1] guidelines in 3.3 (Single Inheritance and Dynamic Dispatch) are easily met when using Ada 2005:

- the *Simple Overriding rule* and the *Simple Dispatch rule* are guaranteed by the language
- the *Accidental Overriding rule* can be met by systematic use of the **`overriding`** keyword (this again can easily be checked with an ASIS tool)
- the *Dispatch Time rule* is a characteristic of the implementation that can be expected to be met provided that the semantics of the language allow constant-time dispatching through compiler generated dispatch tables.

It has often been observed that dispatching calls are equivalent to case statements. This is of course not the case at the conceptual level of the application developer, since the point of using dynamic dispatching is that the programmer invoking an

operation at any particular point does not need to know the set of possible method destinations. In fact nowhere in the entire source set does anyone have to know the set of destinations. That is at once the strength and weakness of the dynamic polymorphism model. The programmer does not know what the flow of control will be, and the flow of control is not known until run time. This is true for all conditional constructs. What distinguishes the dispatching case is that the programmer does not even know statically the set of possible control flows. Since an important element of certification is static testing of all possible control flow paths, there is a fundamental contradiction between the aims and requirements of the programmer at the source level, and the aims and requirements of certification. Dispatching is all about not having to know the possible control flow. Certification is all about needing to know it.

However, once a program has been written, there are indeed transformations that can convert dispatching to the equivalent of case statements. These transformations can be implemented as source-to-source transformations, or can be performed as linker-level activities. We will describe the transformations in terms of source rewrites. The first step is to observe that although during the writing of any particular component of the program, the final set of possible destinations of a dispatching call is unknown, this set is well known by the time the program is linked.

In this discussion we assume that a static linking step produces the executable for a given program. If we consider the realm of highly dynamic object oriented environments which are common with Smalltalk or Java, but can also exist in Ada (see [16]), where new classes can be introduced during the execution of the program, then the situation is much more difficult. We do not attempt to address the issue of certification in such environments. Given that we know the possible set of destinations, the idea is to replace a dynamic call:

```
   Object.Operation;
```

by a proper Ada case statement

```
case Object'Tag'Index is
   when Subtype1'Tag'Index =>
      Subtype1 (Object).Operation;
   when subtype2'Tag'Index =>
      Subtype2 (Object).Operation;
   …
   when others =>
      raise Program_Error with "invalid tag";
end case;
```

Here the calls are not dispatching since the Object is converted to its actual subtype. The set of possible cases is complete since such a transformation is done with a view of the entire program.

There are two ways to replace dispatching calls with case statements. First, we can directly replace each call by such a case statement. Hence, if there are multiple calls to the same operation there will be multiple case statements. Second, we can make a single procedure containing the case statement, and then replace each dynamic call with a call to this procedure.

OOTiA [1], in section 3.12.5, provides a good discussion of the amount of testing required for adequately covering dispatching calls. It proposes four possible levels of coverage. The most stringent is equivalent to traditional covering of the first suggested transformation, that is to say, covering all methods that can be dispatched to at all call points. It is considered a "pessimistic approach" and would probably generate an immense burden on applications that use dispatching extensively. The most optimistic approach is equivalent to the coverage of the second transformation. It is considered to be a relatively weak approach to testing for safety-critical applications. Intermediate approaches based on mathematically significant subsets or equivalence classes amount to selectively using one or the other transformation. At the time of writing, the most fitting approach remains open. Specific guidelines are expected to appear as part of the current work on DO-178C [7].

Note that doing an actual transformation of the code into case statements (rather than just arguing for equivalence at a tool level) offers substantial advantages. First, the usual implementation of dispatching is to issue an indirect call. This means that if the tag of an object is corrupted, we get a worst case erroneous execution involving a random jump. For a case statement, if the tag is corrupted, the effect is much more strictly bounded, since the "when others" branch is taken and a well-known error condition is raised. Second, case statements are known constructs, completely familiar to existing tools for producing certification evidence in non-OO contexts, so the potential for reuse of such toolsets without substantial redesign is attractive.

An enhancement planned for GNAT will implement both schemes for rewriting case statements. The way this will work is that there will be a compiler option to generate appropriate calls, and a binder option to generate the source of the case statements (one per call, or one per operation, depending on the option chosen). These will be generated as normal Ada source and will therefore be fully processable by any normal tools, including the debugger and certification tools. Comments added to generated

case statements will provide traceability back to the original calls in the application source units.

6. What about Multiple Inheritance?

OOTiA's[1] purpose is to define a safe path for the introduction of OO techniques in safety critical software. As shown in previous sections, safety issues related to single inheritance, especially in strongly typed languages, are relatively well understood and are amenable to known and proven approaches. On the other hand, the document is much more circumspect with respect to multiple inheritance and hints several times at doubts as to its usability in a safety-critical context. See, in particular, section 2.3.1.3:

> *…. For example, ANSI C++ has some language features, such as multiple inheritance, that may make it difficult to meet some DO-178B objectives….*

However, OOTiA makes a strong distinction between multiple inheritance of implementation as provided by C++, and delegation, or interface inheritance as is provided by Java and Ada 2005. If the avionics community does not seem ready to go so far as to allow the former, the secure use of the latter does not seem out of reach. See for instance 3.4.3:

Although use of these guidelines helps us deal with the issues raised with respect to ambiguity and complexity, delegation is still considered preferable to the use of multiple implementation inheritance for most systems (as recommended by the AVSI Guide [12]).

One of the major guidelines in [1] is the *repeated interface inheritance rule*, whose intent is to avoid potential ambiguities introduced by inheriting the same operation from different parent interfaces. This condition is guaranteed by Ada 2005 semantics. The other guidelines related to multiple inheritance of interfaces are either not relevant to Ada or can be easily checked by ASIS-based tools such as the ones mentioned in section 2. Therefore, this is another domain where Ada provides what seems to be the appropriate level of support for a feature that may become accepted in safety-critical contexts in the not-too-distant future.

7. Conclusion

The latest revision of the Ada language includes a complete scalable object-oriented model that mixes well with Ada's strong typing tradition. This model is well suited to answer the challenge of introducing various object-oriented techniques into the development of applications requiring various levels of certification. Its most restricted object model can be used with today's most strict certification practices. If dynamic dispatching creates difficulties with current certification practices, Ada is well positioned to offer safe alternatives to overcome those difficulties.

8. Acknowledgement

Many thanks to John Barnes, Kathy Fairlamb, Ed Falis, Jose Ruiz and Ed Schonberg for their invaluable reviews and comments.

9. References

[1] FAA: "*Handbook for Object-Oriented Technology in Aviation (OOTiA)*", FAA 2004, available at http://www.faa.gov/aircraft/air_cert/design_approvals/air_software/oot/.

[2] ISO: "*Ada 95 Reference Manual: Language and Standard Libraries. International Standard*" ANSI/ISO/IEC-8652:1995. Available from Springer-Verlag, LNCS n°1246.

[3] Barnes J.: "*Rationale for Ada 2005*". Draft available at http://www.adacore.com/ada_2005.php

[4] Barnes J.: "*Programming with Ada 95*". Addison-Wesley, 1998. (ISBN 0-201-34293-6)

[5] RTCA: "*Software Consideration in Airborne Systems and Equipment certification*". RTCA/DO-178B, Dec 1rst, 1992.

[6] RTCA: *Final report for clarification of DO-178B: "Software Consideration in Airborne Systems and Equipment Certification".* RTCA/DO-248B, Oct 12, 2001.

[7] RTCA: "SC-205, Software Considerations" (preparation of DO-178C). http://www.rtca.org/comm/Committee.cfm?id=55

[8] Botcazou E., Comar C., Hainque O.: "Compile-Time Evaluation of Stack Size Requirements with GCC", GCC Developer Summit, Ottawa, June 2005.

[9] Dewar R., Hainque O., Craeynest Dirk, Waroquiers P. : "*Exposing Uninitialized Variables : Strengthening and Extending Run-Time Checks in Ada*". http://www.cs.kuleuven.ac.be/~dirk/papers/ae02cfmu-paper.pdf

[10] ISO: "*Guide for the use of the Ada programming language in high integrity systems*". ISO/IEC TR 15942.

[11] Barnes J.: "*High Integrity Software: The SPARK Approach to Safety and Security*", Addison-Wesley, 2003.

[12] AVSI: "*Guide to the Certification of Systems with Embedded Object-Oriented Software*". from the Aerospace Vehicle Systems Institute (AVSI).

[13] AdaCore: "*GNAT Pro User's Guide*".

[14] SIAM: *"Inquiry Board Traces Ariane 5 Failure to Overflow Error"*, http://www.siam.org/siamnews/general/ariane.htm

[15] Comar C., Rogers P.: "*On Dynamic Plug-in Loading with Ada 95 and Ada 2005*", ACM SIGAda Ada letters, Volume 25, Issue 2 (Jun 2005).

[16] ISO: "*Ada Semantic Interface Specification (ASIS)*", ISO/IEC 15291:1999 Information technology -- Programming languages. http://www.iso.org/iso/en/CatalogueDetailPage.CatalogueDetail?CSNUMBER=27169&ICS1=35&ICS2=60&ICS3=

[17] ARG: "*Ravenscar profile for high-integrity systems*", Technical report, ISO/IEC/JTC1/SC22/WG9. Available at http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AIs/AI-00249.TXT

[18] Burns A.: "*The Ravenscar Profile*", Technical report, University of York, 2002. Available at http://www.cs.york.ac.uk/~burns/ravenscar.ps

[19] ACAA: "Ada Conformity Assessment Test Suite (ACATS)", 2005, ACAA. Available at http://www.ada-auth.org/acats.html

10. Glossary

ASIS: Ada Semantic Interface Specification

EUROCAE : European Organization for Civil Aviation Equipment

DO-178B: Software Consideration in Airborne Systems and Equipment certification

DO-248B: Final report for clarification of DO-178B

OO: Object-Oriented

OOTiA: Handbook for Object-Oriented Technology in Aviation

RTCA : Association of aeronautical organizations of the USA from both government and industry.