Towards Certification of Object-Oriented Code with the GNAT Compiler

Javier Miranda

Instituto Universitario de Microelectrónica Aplicada. Universidad de Las Palmas de Gran Canaria, Canary Islands, Spain; email: jmiranda@iuma.ulpgc.es

Courant Institute: Computer Science Department. New York University. 251 Mercer Street, NY 10012.

AdaCore. 104 Fifh Avenue, 15th floor. New York, NY 10011.

Abstract

Dynamic binding, the ability to link at runtime a method call with a subprogram that depends on the class of the object, is strongly discouraged by current standards for avionics airborne systems. This is partly due to dynamic dispatching, the technique commonly used by most OO compilers to implement dynamic binding. In this paper we present some enhancements to the GNAT technology that will help the avionic industry take advantage of the full benefits of the OO techniques with Ada without the inconveniences associated with dynamic dispatching.

Keywords: Dynamic dispatching, Airborne Systems, High-Integrity, Ada 2005, Tagged Types, Abstract Interface Types, GNAT.

1 Introduction

Reliable software construction has evolved considerably in the last two decades. There is currently a trend towards the use of Object Oriented Techniques (OOT) in the construction of High-Integrity Software Systems, such as avionics airborne systems. In this domain, one of the objectives of the forthcoming revision of the DO-178B standard [9] is to address the use of OOT and their associated development processes in the avionics industry. A preliminary document of the future DO-178C [3] provides a comprehensive analysis on the safety concerns associated with OO techniques in the context of DO-178B.

Since the emergence of the DO-178B standard, Ada [11] 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). In recent years Ada has evolved to fulfill the requirements of modern software industry incorporating object-oriented features into its original type model. The Ada 95 standard added to Ada tagged types, single inheritance, polymorphism, and dynamic dispatching. The latest revision of the language, known as Ada 2005, adds multiple inheritance of abstract interface types and numerous other object-oriented programming idioms.

A crucial element of Object Oriented Programming (OOP) is *dynamic binding*, that is the ability to link at runtime a method call with a subprogram based on the class of the object on which the method is invoked. In their current form, DO-178B is wary of dynamic binding: its use is not formally banned, but it is strongly discouraged by DO-248B [10, FAQ 34]. This is partly due to *dynamic dispatching*, the technique used to implement dynamic binding in most compiled OO languages. Although solutions to these issues are emerging, they are not yet fully established.

In this paper we present several ongoing research projects whose main purpose is to facilitate the certification of OO code written in Ada with the GNAT compiler. In Section 2 we summarize inheritance and polymorphism concepts and their common implementation by means of dispatching tables. In Section 3 we describe the main problems of dynamic dispatching in the context of safety and security systems. In Section 4 we present four enhancement projects of the GNAT technology that will help to certify OO Ada code for High-Integrity systems. We close with some conclusions and the bibliography.

2 Inheritance and Polymorphism in Ada

Inheritance was originally viewed as a mechanism for sharing code and data definitions. Multiple inheritance was viewed as a mechanism for constructing a subclass implementation from multiple superclass implementations. As understanding of OO modeling has matured, however, the focus has increasingly been on the specification of interfaces and the specification of interfaces as contracts between clients and implementers. Multiple inheritance is currently used primarily as a means of classifying entities that logically belong to more than a single category. As a result, languages such as Java [5] and Ada 2005 [11] only support multiple inheritance of interfaces and rely on delegation to achieve the effects of multiple implementation inheritance.

In the context of High-Integrity Systems, the general OO avionics guidance [3, Section 3.4] makes a strong distinction between multiple inheritance of specifications and multiple inheritance of implementations as provided by C++, and recommends use of multiple implementation inheritance only for level D software. (The DO-178B

standard defines five levels of safety-criticality, ranging from Level A at the most critical, to Level E at the least critical; the top three safety levels are of particular interest to Ada developers.)

Polymorphism permits instances of a subclass to be assigned to variables of a superclass, and it is used to specify generic algorithms that are common to a given hierarchy of classes. In this context, dynamic binding ensures that the method executed by a call to a polymorphic object is that associated with the object's run time type. Conceptually, at run-time there is a single dispatch routine containing a pair of nested case statements [3, Section 3.3.2]:

```
case <Object'Run-Time-Type> is
   ...
   when <Class-N> =>
     case <Method'Signature> is
       ...
       when <Method-N> =>
         call <Method-N> defined by <Class-N>
      end case
end case
```

In practice, dynamic binding is typically implemented using *Dispatch Tables*, which introduce a small and fixed overhead (cf. Figure 1). Each object instance has a hidden component (the *Vtable* pointer in C++ and Java, or the *Tag* in Ada) that references a dispatch table with the run-time type's method signatures. (For efficiency reasons the method signature is generally the address of the target method.) 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.

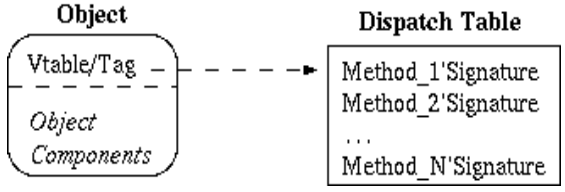

Figure 1 Object Layout

In Ada, 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 primitive 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 (*T'Class* denotes the entire set of types in the class of *T*). This flexibility can be used to limit the number of dispatching calls, thereby limiting their associated certification cost. Prevention of dispatching can be also enforced by the use of pragma Restrictions (No_Dispatch).

This flexibility is not available in Java where all operation invocations are dispatching (unless a routine is declared as final, which allows the compiler to perform various optimizations knowing the primitive cannot be overridden). 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++ compilers are allowed to optimize dispatching calls into regular calls when the context permits, but this is not under the control of the developer.

3 Problems with Dynamic Dispatching in High-Integrity Systems

Dynamic dispatching has several safety and security problems, namely:

- **Initialization**: how can we prove that dispatch tables and *Tag* fields are initialized correctly?
- **Modification**: how can we prove that dispatch table and *Tag* values are not updated maliciously or unintentionally during the execution of a program?
- **Tools**: being dynamic dispatching invisible at the source level, how can we use source-based tools in the presence of dynamic dispatching for code coverage?

Demonstrating correct dispatch-table initialization at object-level is akin to the problem of showing that the linker produces a correct executable from the object files it links. This problem is part of the control coupling objective in DO-178B parlance and is addressed by either verifying the correctness of the final result by hand of by employing a qualified tool that performs such verification [12].

If one can ROM dispatch tables or place them in OS-guarded read-only memory the need to verify that dispatch tables are unchanged during a program's execution disappears. Unfortunately, an object's *Tag* field cannot typically be placed into read-only memory and the costs of demonstrating at object-code level that these fields are unchanged during a program's execution remain. Such modifications could occur because of a rogue pointer or buffer overflow in assembly or C/C++ code that may be part of the application or by other accidental or malicious means.

In the following section we present several enhancement projects that will help to workaround these problems with the GNAT technology.

4 Towards certification of dispatching calls with the GNAT compiler

In order to certify dispatching calls in High-Integrity Software the following concerns of the general OO avionics guidance must be answered by the compiler [3]:

- **Stack Analysis**: ``Stack overflow errors are listed in section 6.4.3f of DO-178B as errors that are typically found in requirements-based hardware-software integration testing. Timing and stack

analysis are complicated by certain implementations of dynamic dispatch. If polymorphism and dynamic binding are implemented, stack size can grow, making analysis of the optimal stack size difficult" [3, Section 2.3.3.1.3].

- **Object Code Traceability**: ``Everywhere concerns about source code to object code traceability and timing analysis dictate, the compiler vendor may be asked to provide evidence of deterministic, bounded mapping of the dispatched call. If the evidence is not available from the compiler vendor, it may be necessary to examine the structure of the compiler-generated code and data structures (e.g., method tables) at the point of call" [3, Section 3.3.4.3].
- **Structural Coverage**: ``Many current Structural Coverage Analysis tools do not ``understand" dynamic dispatch, i.e. do not treat it as equivalent to a call to a dispatch routine containing a case statement that selects between alternative methods based on the run-time type of the object. (IL 55)" [3, Section 2.3.3.1.2].

In this section we present several ongoing enhancements to the GNAT technology that will help to solve these problems. The concern of stack-analysis has been already addressed by GNAT with the gnatstack tool (work described in a separate paper [1]). In the following sections we present three additional enhancement projects: in Section 4.1 we present the visualization of the dispatch tables at the source level; in Section 4.2 we present the static allocation of dispatch tables. These projects are currently at their final stage. Finally in Section 4.3 we present a new project that expands dispatching calls into case statements.

4.1 Dispatch Table Visualization

The first enhancement of the GNAT technology addresses the correct initialization of the dispatch table. The compiler has being improved to leave the initialization of the dispatch tables visible at source level and hence to support the DO-178B traceability requirement. Using a switch the compiler currently generates Ada-like code that allows to see the expansion performed by the frontend. As part of this project, the output associated with the construction of dispatch tables has been improved to facilitate the use of source-based tools based on static control flow to verify their correct initialization. Such Ada-like code can be also visualized during debugging using another compiler switch. Let us consider the following Ada 2005 example to present this new output:

```
package Iface is
   type Writable is interface;
   procedure Read
      (Obj : Writable; Data : out Integer) is abstract;
   procedure Write
      (Obj : Writable; Data : in Integer)  is abstract;
end Iface;
```

Package Iface contains the declaration of the abstract interface type Writable that has two abstract primitives: Read and Write.

```
with Iface; use Iface;
package Pkg is
   type Root is tagged ... ;
   function Is_Empty (Obj : Root) return Boolean;

   type Derived is new Root and Writable with ...;
   procedure Read  (Obj : Derived; Data : out Integer);
   procedure Write (Obj : Derived; Data  : in  Integer);
end Pkg;
```

Package Pkg defines the root of derivation of a tagged type in which all descendants have the primitive operation Is_Empty. The package also has a derivation of Root that acquires the obligation of implementing all the primitives of interface Writable. Figure 2 presents the layout of an object of type Derived and its GNAT run-time structure.

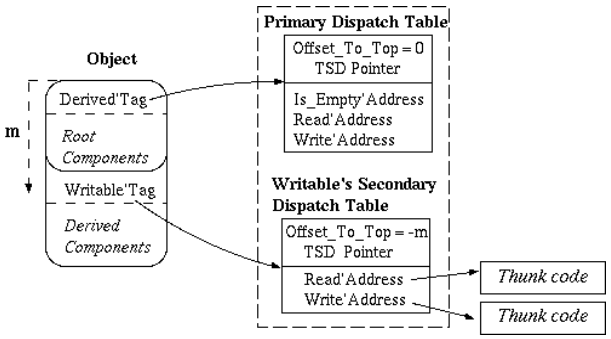

Figure 2 GNAT Object Layout

Each tagged type has one primary dispatch table, associated with its main root of derivation, plus one secondary dispatch table associated with each progenitor (a progenitor is one of the types given in the definition of a derived type other than the parent type ---AARM Annex N). In our example, each object of type Derived has one primary dispatch table plus one secondary dispatch table associated with the interface type Writable. Each dispatch table has a header containing the offset to the top and the pointer to the *Run-Time Type Specific Data* record (TSD). For a primary dispatch table, the Offset_To_Top component is always set to 0; for secondary dispatch tables the Offset_To_Top component holds the displacement to the top of the object from the object component containing the interface tag (in the figure the value of this offset is *m*). After the TSD component the dispatch tables have the table of pointers to primitive operations. In secondary dispatch tables, rather than containing direct pointers to the primitives associated with the interfaces, the dispatch table contain pointers to small fragments of code called *thunks*. These thunks are used to adjust the pointer to the base of the object during interface type conversions. For further information on the object layout and the GNAT run-time structures associated with interface types read [6, 7, 8]).

In order to present the new output associated with the construction of the dispatch table of type Derived let us compile the above Ada example using switch –gnatD:

```
Derived__ID :                                          -1-
  aliased constant string := "PKG.DERIVED";

Derived__Ifaces :                                      -2-
  aliased constant Interface_Data (1) :=
    (Num_Ifaces    => 1,
     Ifaces_Tables => (Derived__Writable_Tag));

Derived__TSD :                                         -3-
  aliased constant Type_Specific_Data (Idepth => 1) :=
    (Idepth              => 1,
     Access_Level        => 0,
     Expanded_Name       => Derived__ID'Address,
     External_Tag        => Derived__ID'Address,
     HT_Link             => null,
     Transportable       => False,
     RC_Offset           => 0,
     Interfaces_Table    => Derived_Ifaces'Address,
     SSD                 => null,
     Tags_Table          => (Derived_Tag, Root_Tag));

Derived__Predef_Prims :                                -4-
  aliased constant Address_Array (1 .. 10) :=
    (1  => Derived__Size'Address,
     ...
     10 => Root__DF'Address);

Derived__DT :                                          -5-
  aliased constant Dispatch_Table (Num_Prims => 3) :=
    (Num_prims      => 3,
     Signature      => Primary_DT,
     Tag_kind       => TK_tagged,
     Predef_prims   => Derived__Predef_Prims'Address,
     Offset_to_top  => 0,
     TSD            => Derived__TSD'Address,
     Prims_Ptr => (
       1 => Is_Empty'Address,
       2 => Read'Address,
       3 => Write'Address));

Derived__Tag :                                         -6-
  constant Tag := Derived__DT.Prims_Ptr'Address;

Register_Tag (Derived__Tag);                           -7-
```

At -1- we see the declaration of an object containing the external tag of Derived; at -2- we find the declaration and initialization of a table containing the tags of all the interfaces covered by Derived (in this example, just one); at -3- we have the Run-Time Type Specific Data record of Derived; at -4- we see the dispatch table of its predefined primitives; at -5- we see the primary dispatch table associated with Derived; at -6- we find the declaration of the Tag associated with this primary dispatch table (a copy of this tag will be saved in the _Tag component of objects of type Derived during their initialization); finally at -7- we find the code that registers the tags in the run-time (required to support the Internal_Tag service of standard Ada package Ada.Tags). For further information on the contents of each component see the documentation available in the source of *a-tags.ads*.

The expansion of dispatching calls makes use of the tag of the object and the compile-time known position of the target primitive to index the Prims_Ptr element containing the pointer to the target primitive. That is, considering the following example, in the commented line we see the expansion of the dispatching call to Is_Empty.

```
function Dispatch_Test (Obj : Root'Class)
  return Boolean is
begin
   return Obj.Is_Empty;
   -- Expanded into: return obj._Tag (1).all (obj);
end Dispatch_Test;
```

Source-based tools can use this new output to verify the correct construction of the dispatch table; they should check the subprograms referenced in the aggregates that initialize the dispatch table associated with predefined primitives (*Predef_Prims*) and the dispatch table containing the user-defined primitives (*Prims_Ptr*). For this purpose the compiler generates unique names for all subprograms found in the sources (including overloaded subprograms).

4.2 Static Allocation of Dispatch Tables

Another enhancement of the GNAT compiler is the improvement of its code generation to statically allocate dispatch tables associated with tagged types defined at the library level. In order to present it let us see the assembly code generated by GNAT when compiling the previous example for i86 architectures. For this purpose we compile our example using two additional switches (*-fverbose-asm* and *-save-temps*). The following fragment of assembly code corresponds to the declaration and initialization of the dispatch table containing the predefined primitives (object declaration found at -4- of previous Section):

```
pkg__derived_dt:
        .long pkg___size__2
        .long pkg___alignment__2
        .long pkg__derivedSR
        .long pkg__derivedSW
        .long pkg__derivedSI
        .long pkg__derivedSO
        .long pkg__Oeq__2
        .long pkg___assign__2
        .long pkg__rootDA
        .long pkg__rootDF
```

As the reader can see, the compiler generates external symbols for the table entries, rather than relying on the generation of run-time code to initialize table entries with addresses of code. For certification purposes, this is a major improvement in the code generation; previous versions of the compiler declare dispatch tables as un-initialized objects that are initialized during the elaboration of the package by means of additional assignments generated by

the compiler. Combined with GNAT specific pragmas this new feature allows placement of dispatch tables in ROM or in OS-guarded read-only memory.

4.3 Transformation of dynamic dispatching call into case-statement

Another concern for certifying OO code in HI software is the compiler support for Structural Coverage Analysis tools. The DO-178B stablishes three kinds of coverage requirements: Level C specifies *Statement Coverage*, which requires every statement in the program to have been invoked at least once. Level B specifies *Decision Coverage*, which requires every point of entry and exit in the program has been invoked at least once and every decision in the program has taken on all possible outcomes at least once. Finally, level A requires *Modified Condition/Decision Condition* (MCDC) testing, which involves testing all the permutations of conditions involving several logic operators. Dynamic dispatch complicates flow analysis of coverage requirements because reading the sources is it unclear which method in the inheritance hierarchy will be called [3, Section 2.2.3.1.1].

In order to help certifying Level A software with Ada, we are enhancing GNAT to expand dispatching calls into the equivalent case statements [2, 4]. The key point of this project is the following observation: 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 at link-time (we assume that a static linking step produces the executable for a given program). Therefore, instead of generating the usual transformation for a dispatching call, at the point of the call the GNAT compiler will generate the following code:

```
R := Obj.Is_Empty;
-- Expanded into: Find_Method (Obj, Object.Tag);
```

The post-processing part of the code transformation is performed at bind-time. This involves generating the body for routine Find_Method which implements dynamic binding with an explicit case statement as shown bellow:

```
procedure Find_Method
  (Obj : Root'Class; The_Tag : Positive) is
begin
   case The_Tag is
      when Root'Tag =>
         Root (Object).Is_Empty;
      when Derived'Tag =>
         Derived (Object).Is_Empty;
      ...
    end case;
end Find_Method;
```

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

The following implementation model is underway: a compiler option prevents the dispatching expansion described earlier, and a separate switch forces the binder to generate the source code for the case statements. This is legal Ada source code, which is therefore fully processable by standard tools, including the debugger and certification tools.

5 Conclusions

Ada is clearly a safe and efficient vehicle to create certifiable systems. It has been used successfully in many major aeronautics projects (Boeing 777, A340, and more recently Boeing 787, A380 and A400M). In the recent years Ada has evolved to fulfill the requirements of modern software industry incorporating object-oriented features into its original type model. The Ada 95 standard added to Ada tagged types, single inheritance, polymorphism, and dynamic dispatching. The latest revision of the language, informally known as Ada 2005 [11], adds multiple inheritance of abstract interface types and numerous other object-oriented programming idioms.

A preliminary version of the incoming DO-178C standard for avionics provides a comprehensive analysis on safety concerns associated with OO techniques in the context of DO-178B. Such document states that dispatching calls (the technique commonly used by most compilers for OO languages) is clearly unacceptable in this context. In this paper we have presented some enhancement projects of the GNAT technology that will help the industry to take advangage of the full benefits of the OO techniques with Ada without the inconveniences associated with dynamic dispatching, namely:

- **Dispatch table visualization**. Enhancement that modifies the compiler to make the initialization of the dispatch tables visible at the source level. In addition, the code generated by the compiler will be also visualized during debugging using another compiler switch. This project gives support to DO-178B traceability requirements.
- **Static allocation of dispatch tables**. Enhancement that improves the code generation of the compiler to allow the static allocation of dispatch tables associated with tagged types defined at the library level. This project will allow the placement of the dispatch tables in ROM or in OS-guarded read-only memory.
- **Translation of dispatching call into case-statement**. Enhancement that modifies the compiler to expand dispatching calls into the equivalent case statements. This project gives support to the structural coverage analysis and verification for level A systems as dictated by DO-178B.
- **Stack analysis tool**. This enhancement is already finished, and the gnatstack tool is currently part of the GNAT Pro toolset [1].

Since the emergence of the DO-178B standard, Ada has been one of the few languages of choice for the construction of HI systems. We expect that these enhancement projects to the GNAT technology will help Ada to keep this leadership.

Acknowledgements

Most of this work was done during a six-month visit to the NYU Courant Institute funded by the Spanish Minister of Education and Science under project PR2006-0356.

I give special thanks to professor Edmond Schonberg for his continuous help and support, and Professor Robert Dewar not only for the technical discussions but also for his hospitality. I also acknowledge the contributions of Cyrille Comar, Franco Gasperoni, Richard Kenner, and Eric Botcazou that helped me to go ahead with this work.

References

[1] E. Botcazou, C. Comar, and O. Hainque (2005), *Compile-time stack requirements analysis with GCC*, June, 2005. Available at: http://www.adacore.com/2005/06/01/compile-time-stack-requirements-analysis-with-gcc/

[2] C. Comar, R. Dewar, and G. Dismukes (2006), *Certification & Ob ject Orientation: The New Ada Answer*, March, 2006. Available at: http://www.adacore.com/2006/03/08/certification-object-orientation-the-new-ada-answer/

[3] FAA (2004), *Handbook for Object-Oriented Technology in Aviation* (OOTiA), October 26, 2004. Available at: http://www.faa.gov/aircraft/air_cert/design_approvals/air_software/oot/

[4] F. Gasperoni (2006), *Safety, Security, and Object-Oriented Programming,* March, 2006. Available at: http://www.adacore.com/2006/03/30/safety-security-and-object-oriented-programming/

[5] J. Gosling, B. Joy, G. Steele, and G. Bracha (2000), *The Java Language Specification*, 2nd edition, Addison-Wesley.

[6] J. Miranda, E. Schonberg, and G. Dismukes (2005), *The Implementation of Ada 2005 Interface Types in the GNAT Compiler*, 10th International Conference on Reliable Software Technologies, Ada-Europe 2005, LNCS 3555, pp. 208–219, Springer-Verlag.

[7] J. Miranda, Schonberg E., and G. Dismukes (2006), *Abstract Interface Types in GNAT: Conversions, Discriminants, and C++*, 11th International Conference on Reliable Software Technologies, Ada-Europe 2006, LNCS 4006, pp. 179–190, Springer-Verlag.

[8] J. Miranda, Schonberg E., and H. Kirtchev (2005), *The Implementation of Ada 2005 Synchronized Interfaces in the GNAT Compiler*, Proceedings of the 2005 annual ACM SIGAda International Conference on Ada, pp. 41–48.

[9] RTCA (1992), *Software Consideration in Airborne Systems and Equipment Certification*, RTCA/DO-178B, December, 1992.

[10] RTCA (2001), *Final report for clarification of DO-178B: Software Consideration in Airborne Systems and Equipment Certification*, RTCA/DO-248B, October, 2001.

[11] S. T. Taft, R. A. Duff, R. L. Brukardt, E. Plödereder, P. Leroy (eds) (2007), *Ada 2005 Reference Manual: Language and Standard Libraries International Standard ISO/IEC 8652:1995(E) with Technical Corrigendum 1 and Ammendment 1*, LNCS 4348, Springer-Verlag. ISBN: 3-540-69335-1.

[12] VEROCEL (2006), *VerOLink: Verify Object Linking*, http://www.verocel.com/verolink.htm.