Abstract Interface Types in GNAT: Conversions, Discriminants, and C++

Javier Miranda[1] and Edmond Schonberg[2]

[1] jmiranda@iuma.ulpgc.es
Applied Microelectronics Research Institute
University of Las Palmas de Gran Canaria
Spain
and AdaCore

[2] schonberg@adacore.com
AdaCore
104 Fifth Avenue, 15th floor
New York, NY 10011

Abstract. Ada 2005 Abstract Interface Types provide a limited and practical form of multiple inheritance of specifications. In this paper we cover the following aspects of their implementation in the GNAT compiler: interface type conversions, the layout of variable sized tagged objects with interface progenitors, and the use of the GNAT compiler for interfacing with C++ classes with compatible inheritance trees.

Keywords: Ada 2005, Abstract Interface Types, Tagged Types, Discriminants, GNAT.

1 Introduction

In recent years, a number of language designs [1, 2] have adopted a compromise between full multiple inheritance and strict single inheritance, which is to allow multiple inheritance of *specifications*, but only single inheritance of *implementations*. Typically this is obtained by means of *"interface"* types. An interface consists solely of a set of operation specifications: it has no data components and no operation implementations. A type may implement multiple interfaces, but can inherit code from only one parent type [4, 7]. This model has much of the power of full-blown multiple inheritance, without most of the implementation and semantic difficulties that are manifest in the object model of C++ [3].

At compile time, an interface type is conceptually a special kind of *abstract tagged type* and hence its handling does not add special complexity to the compiler front-end (in fact, most of the current compiler support for abstract tagged types has been reused in GNAT). At run-time we have chosen to give support to dynamic dispatching through abstract interfaces by means of secondary dispatch tables. This model was chosen for its time efficiency (constant-time dispatching

through interfaces), and its compatibility with the run-time structures used by G++ (this is the traditional nickname of GNU C++).

This is the third paper in a series describing the implementation in the GNAT compiler of Ada 2005 features related to interfaces (the previous papers are [13] and [14]). We discuss the interesting implementation challenges presented by interface type conversions, and the layout of variable sized tagged objects with *progenitors*, which is the Ada 2005 term that designates the interfaces implemented by a tagged type [4, Section 3.9.4 (9/2)]. Finally, we show how our implementation makes it possible to write multi-language object-oriented programs with interface inheritance. We present a small mixed-language example that imports into Ada a C++ class hierarchy with multiple inheritance, when all but one of the *base* classes have only *pure virtual* functions [3, Chapter 9].

This paper is structured as follows: In Section 2 we give a brief overview of Ada 2005 abstract interfaces. In Section 3 we summarize the data layout adopted by GNAT (for more details read [13] and [14]). In Section 4 we describe the implementation of interface conversions. In Section 5 we discuss possible approaches to the layout of tagged objects with components constrained by discriminants, and their impact on conversion. In Section 6 we present an example of mixed-language object-oriented programming; this example extends in Ada 2005 a C++ class whose *base* classes have only *pure virtual functions* [3, Chapter 9]. We close with some conclusions and the bibliography.

2 Abstract Interfaces in Ada 2005

The characteristics of an Ada 2005 interface type are introduced by means of an interface type declaration and a set of subprogram declarations [4, Section 3.9.4]. The interface type has no data components, and its primitive operations are either abstract or null. A type that implements an interface must provide non-abstract versions of all the abstract operations of its *progenitor(s)*. For example:

```
package Interfaces_Example is
   type I1 is interface;                                    -- 1
   function P (X : I1) return Natural is abstract;

   type I2 is interface and I1;                             -- 2
   procedure Q (X : I1) is null;
   procedure R (X : I2) is abstract;

   type Root is tagged record ...                           -- 3
   ...
   type DT1 is new Root and I2 with ...                     -- 4
   --  DT1 must provide implementations for P and R
   ...
   type DT2 is new DT1 with ...                             -- 5
   --  Inherits all the primitives and interfaces of the ancestor
end Interfaces_Example;
```

The interface I1 defined at –1– has one subprogram. The interface I2 has the same operations as I1 plus two subprograms: the null subprogram *Q* and the abstract subprogram *R*. (Null procedures are described in AI-348 [10]; they behave as if their body consists solely of a *null_statement*.) At –3– we define the root of a derivation class. At –4– *DT1* extends the root type, with the added commitment of implementing (all the abstract subprograms of) interface I2. Finally, at –5– type *DT2* extends DT1, inheriting all the primitive operations and interfaces of its ancestor.

The power of multiple inheritance is realized by the ability to dispatch calls through interface subprograms, using a controlling argument of a class-wide interface type. In addition, languages that provide interfaces [1, 2] provide a run-time mechanism to determine whether a given object implements a particular interface. Accordingly Ada 2005 extends the membership operation to interfaces, and allows the programmer to write the predicate *O in I'Class*. Let us look at an example that uses the types declared in the previous fragment, and displays both of these features:

```
procedure Dispatch_Call (Obj : I1'Class) is
begin
  if Obj in I2'Class then   -- 1: membership test
    R (I2'Class (Obj));     -- 2: interface conversion plus dispatch call
  else
    ... := P (Obj);         -- 3: dispatch call
  end if;

  I1'Write (Stream, Obj);   -- 4: dispatch call to predefined op.
end Dispatch_Call;
```

The type of the formal *Obj* covers all the types that implement the interface I1. At –1– we use the membership test to check if the actual object also implements I2. At –2– we perform a conversion of the actual to the class-wide type of I2 to dispatch the call through I2'Class. (If the object does not implement the target interface and we do not protect the interface conversion with the membership test then Constraint_Error is raised at run-time.) At –3– the subprogram safely dispatches the call to the *P* primitive of I1. Finally, at –4– we see that, in addition to user-defined primitives, we can also dispatch calls to predefined operations (that is, *'Size*, *'Alignment*, *'Read*, *'Write*, *'Input*, *'Output*, *Adjust*, *Finalize*, and the equality operator).

Ada 2005 extends abstract interfaces for their use in concurrency: an interface can be declared to be a non-limited interface, a limited interface, a synchronized interface, a protected interface, or a task interface [9, 11]. Each one of these imposes constraints on the types that can implement such an interface: a task interface can be implemented only by a task type or a single task; a protected interface can only be implemented by a protected type or a single protected object; a synchronized interface can be implemented by either task types, single tasks, protected types or single protected objects, and a limited interface can

be implemented by tasks types, single tasks, protected types, single protected objects, and limited tagged types.

The combination of the interface mechanism with concurrency means that it is possible, for example, to build a system with distinct server tasks that provide similar services through different implementations, and to create heterogeneous pools of such tasks. Using synchronized interfaces one can build a system where some coordination actions are implemented by means of active threads (tasks) while others are implemented by means of passive monitors (protected types). For details on the GNAT implementation of synchronized interfaces read [14].

3 Abstract Interfaces in GNAT

Our first design decision was to adopt as much as possible a dispatching model compatible with the one used by G++, in the hope that mixed-language programming would intermix types, classes, and operations defined in both languages. A compatible design decision was to ensure that dispatching calls through either classwide types or interface types should take constant time.

As a result of these choices, the GNAT implementation of abstract interfaces is compatible with the C++ Application Binary Interface (ABI) described in [6]. That is, the compiler generates a secondary dispatch table for each progenitor of a given tagged type. Thus, dispatching a call through an interface has the same cost as any other dispatching call. The model incurs storage costs, in the form of additional pointers to dispatch tables in each object.

Figure 1 presents an example of this layout. The dispatch table has a header containing the offset to the top and the Run-Time Type Information Pointer (RTTI). For a primary dispatch table, the first field is always set to 0 and the RTTI pointer points to the GNAT *Type Specific Data* (the contents of this record are described in the GNAT sources, file a-tags.adb). The tag of the object points to the first element of the table of pointers to primitive operations. At the bottom of the same figure we have the layout of a derived type that implements two interfaces I1 and I2. When a type implements several interfaces, its run-time data structure contains one primary dispatch table and one secondary dispatch table per interface. In the layout of the object (left side of the figure), we see that the derived object contains all the components of its parent type plus 1) the tag of all the implemented interfaces, and 2) its own user-defined components. Concerning the contents of the dispatch tables, the primary dispatch table is an extension of the primary dispatch table of its immediate ancestor, and thus contains direct pointers to all the primitive subprograms of the derived type. The *offset_to_top* component of the secondary tables holds the displacement to the top of the object from the object component containing the interface tag. (This offset provides a way to find the top of the object from any derived object that contains secondary virtual tables and is necessary in abstract interface type conversion; this will be described in Section 4.)

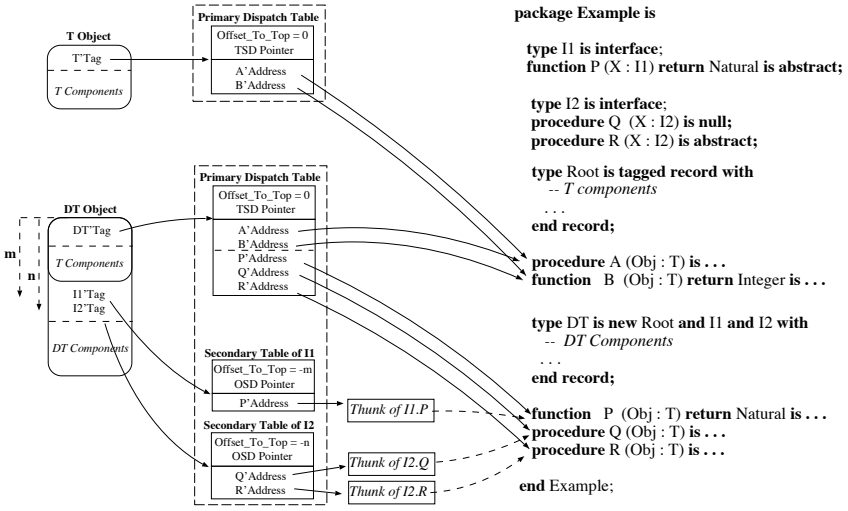

Fig. 1. Layout compatibility with C++

In the example shown in Figure 1, the offset-to-top values of interfaces I1 and I2 are m and n respectively. In addition, rather than containing direct pointers to the primitive operations associated with the interfaces, the secondary dispatch tables contain pointers to small fragments of code called *thunks*. These thunks are generated by the compiler, and used to adjust the pointer to the base of the object (see description below).

4 Abstract Interface Type Conversions

In order to support interface conversions and the membership test, the GNAT run-time has a table of interfaces associated with each tagged type containing the tag of all the implemented interfaces plus its corresponding offset-to-top value in the object layout. Figure 2 completes the run-time data structure described in the previous section with the *Type Specific Data* record which stores this table of interfaces.

In order to understand the actions required to perform interface conversions, let us recall briefly the use of this run-time structure for interface calls. At the point of call to a subprogram whose controlling argument is a class-wide interface, the compiler generates code that displaces the pointer to the object by m bytes, in order to reference the tag of the secondary dispatch table corresponding to the controlling interface. This adjusted address is passed as the pointer to the actual object in the call. Within the body of the called subprogram, the dispatching call to P is handled as if it were a normal dispatching call. For example, because P is the first primitive operation of the interface I1, the compiler generates code that issues a call to the subprogram identified by the first entry

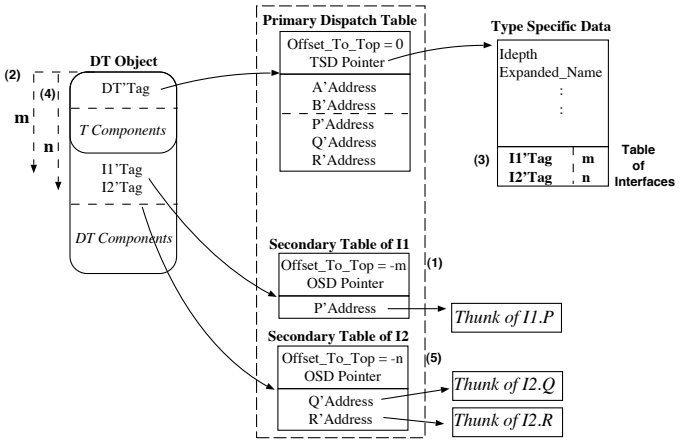

Fig. 2. Object Layout

of the primary dispatch table associated with the actual parameter. Because the actual parameter is a displaced pointer that points to the I1'Tag component of the object, we are really issuing a call through the secondary dispatch table of the object associated with the interface I1. In addition, rather than a direct pointer to subprogram Q, the compiler also generates code that fills this entry of the secondary dispatch table with the address of a *thunk* that 1) subtracts the m bytes displacement corresponding to I1 in order to adjust the address so that it refers to the real base of the object, and 2) does a direct jump to the body of subprogram Q.

Now let us see the work performed by the GNAT run-time to support interface conversion. Let us assume that we are again executing the body of the subprogram with the class-wide formal, and hence that the actual parameter is a displaced pointer that points to the I1'Tag component of the object. In order to get access to the table of interfaces the first step is to read the value of the offset-to-top field available in the header of the dispatch table (see 1 in the figure). This value is used to displace upwards the actual parameter by m bytes to designate the base of the object (see 2). From here we can get access to the table of interfaces and retrieve the tag of the target interface (see 3). If found we perform a second displacement of the actual by using the offset value stored in the table of interfaces (in our example n bytes) to displace the pointer downwards from the root of the object to the component that has the I2'Tag of the object (see 4). If the tag of the target interface is not found in the table of interfaces the program must raise Constraint_Error. As a result, an interface conversion incurs a run-time cost proportional to the number of interfaces implemented by the type. An extensive examination of the Java libraries indicates that in the great majority of cases there are no more than 4 progenitors for any given class. Thus this overhead is certainly acceptable. More sophisticated structures could be used to speed up the search for the desired interface, but we

defer such optimizations until actual performance results indicate that they are needed.

5 Discriminant Complications

The use of abstract interface types in variable sized tagged objects requires some special treatment. Complications arise when a tagged type has a parent that includes some component whose size is determined by a discriminant. For example:

```
type Root (D : Positive) is tagged record
   Name : String (1 .. D);
end record;

type DT is new Root and I1 and I2 with ...
Obj : DT (N);   --  N is not necessarily static
```

In this example it is clear that the final position of the components containing the tags associated with the secondary dispatch tables of the progenitors depends on the actual value of the discriminant at the point the object Obj is elaborated. Therefore the offset-to-top values can not be placed in the header of the secondary dispatch tables, nor in the table of interfaces itself. However as we described in the previous section the offset-to-top values are required for interface conversions. The C++ ABI does not address this problem for the simple reason that C++ classes do not have non-static components.

At this point it is clear that we must provide a way to 1) displace the pointer up to the base of the object, and 2) displace the pointer down to reference the tag component associated with the target interface. Two main alternatives were considered to solve this problem (obviously the naive approach of generating a separate dispatch table for each object was declared unacceptable at once). Whatever alternative was chosen, it should not affect the data layout when discriminants are not present, so as to maintain C++ compatibility for the normal case. The two plausible alternatives are:

1. To place the interface tag components at negative (and static) offsets from the object pointer (cf. Figure 3). Although this solution solves the problem, it was rejected because the value of the *Address* attribute for variable size tagged objects would not be conformant with the Ada Reference Manual, which explicitly states that *"X'Address denotes the address of the first of the storage elements allocated for X"* [5, Annex K]. In addition, programmers generally assume that the allocation of an object can be accurately described using *'Address* and *'Size* and therefore they generally expect to be able to place the object at the start of a particular region of memory by using an offset of zero from that starting address.

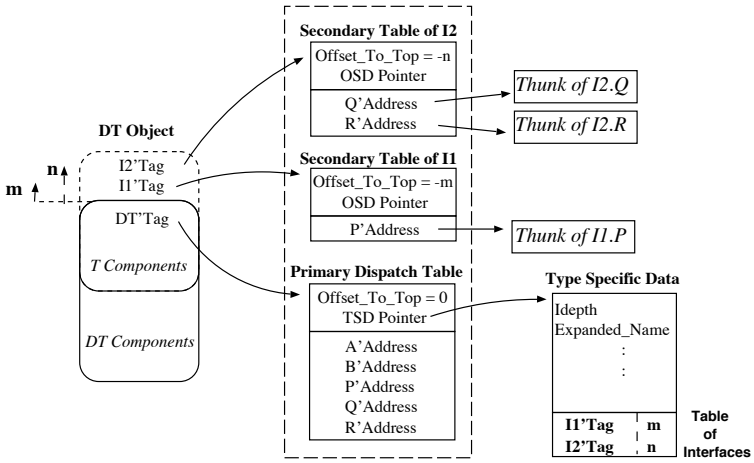

Fig. 3. Approach I: Interface tags located at negative offsets

2. The second option is to store the offset-to-top values immediately following each of the interface tags of the object (that is, adjacent to each of the object's secondary dispatch table pointers). In this way, this offset can be retrieved when we need to adjust a pointer to the base of the object. There are two basic cases where this value needs to be obtained: 1) The thunks associated with a secondary dispatch table for such a type must fetch this offset value and adjust the pointer to the object appropriately before dispatching a call; 2) Class-wide interface type conversions need to adjust the value of the pointer to reference the secondary dispatch table associated with the target type. In this second case this field allows us to solve the first part of the problem, but we still need this value in the table of interfaces to be able to displace down the pointer to reference the field associated with the target interface. For this purpose the compiler must generate object specific functions which read the value of the offset-to-top hidden field. Pointers to these functions are themselves stored in the table of interfaces.

The latter approach has been selected for the GNAT compiler. Figure 4 shows the data layout of our example following this approach. Note: The value -1 in the *Offset_To_Top* of the secondary dispatch tables indicates that this field does not have a valid offset-to-top value.

6 Collaborating with C++

The C++ equivalent of an Ada 2005 abstract interface is a class with pure virtual functions and no data members. For example, the following declarations are conceptually equivalent:

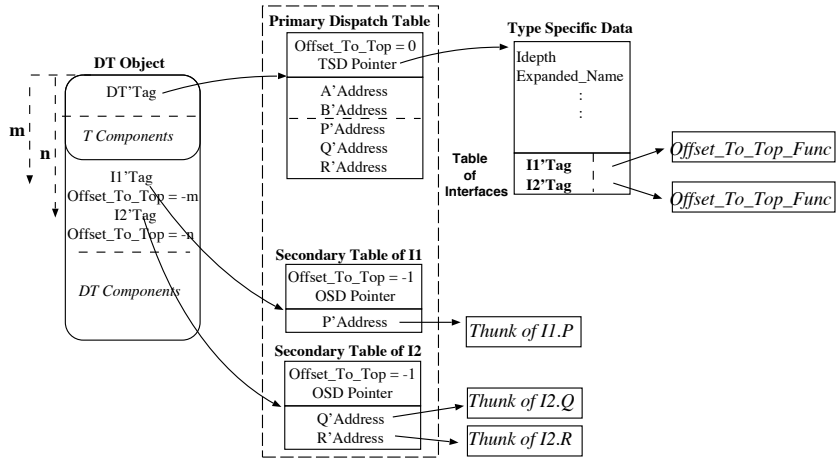

Fig. 4. Approach II: Offset value adjacent to pointer to secondary DT

```
class I1 {                              type I1 is interface;
public: virtual void p () = 0;          procedure P (X : I1) is abstract;
}

class I2 {                              type I2 is interface;
public: virtual void q () = 0;          procedure Q (X : I2) is abstract;
        virtual int  r () = 0;          function  R (X : I2) return Integer
}                                          is abstract;
```

Let us see the correspondence between classes derived from these declarations in the two languages:

```
class Root {                            type Root is tagged record with
public:                                    R_Value : Integer;
  int r_value;                          end record;
  virtual void Root_Op ();              procedure Root_Op (X : Root);
};

class A : Root, I1, I1 {                type A is new Root and I1 and I2 with
public                                     A_Value: Float;
  float a_value;                        end record;

  virtual void  p ();                   procedure P (X : A);
  virtual void  q ();                   procedure Q (X : A);
  virtual int   r ();                   function  R (X : A) return Integer;
  virtual float s ();                   function  S (X : A) return Float;
};
```

Because of the chosen compatibility between GNAT run-time structures and the C++ ABI, interfacing with these C++ classes is easy. The only require-

ment is that all the primitives and components must be declared exactly in the same order in the two languages. The code makes use of several GNAT-specific pragmas, introduced early in our Ada 95 implementation for the more modest goal of using single inheritance hierarchies across languages. These pragmas are CPP_Class, CPP_Virtual, CPP_Import, and CPP_Constructor.

First we must indicate to the GNAT compiler. by means of the pragma CPP_-Class, that some tagged types have been defined in the C++ side; this is required because the dispatch table associated with these tagged types will be built on the C++ side and therefore it will not have the Ada predefined primitives. (The GNAT compiler then generates the elaboration code for the portion of the table that holds the predefined primitives: Size, Alignment, stream operations, etc). Next, for each user-defined primitive operation we must indicate by means of pragma CPP_Virtual that their body is on the C++ side, and by means of pragma CPP_Import their corresponding C++ external name. The complete code for the previous example is as follows:

```
package My_Cpp_Interface is
   type I1 is interface;
   procedure P (X : I1) is abstract;

   type I2 is interface;
   procedure Q (X : I1) is abstract;
   function R (X : I2) return Integer is abstract;

   type Root is tagged record with
      R_Value : Integer;
   end record;
   pragma CPP_Class (Root);

   procedure Root_Op (Obj : Root);
   pragma CPP_Virtual (Root_Op);
   pragma Import (CPP, Root_Op, "_ZN4Root7Root_OpEv");

   type A is new Root and I1 and I2 with record
      A_Value : Float;
   end record;
   pragma CPP_Class (A);

   procedure P (Obj : A);
   pragma CPP_Virtual (P);
   pragma Import (CPP, P, "_ZN1A4PEv");

   procedure Q (Obj : A);
   pragma CPP_Virtual (Q);
   pragma Import (CPP, Q, "_ZN1A4QEv");

   function R (Obj : A) return Integer;
   pragma CPP_Virtual (R);
```

```
   pragma Import (CPP, R, "_ZN1A4REv");

   function S (Obj : A) return Float;
   pragma CPP_Virtual (S);
   pragma Import (CPP, S, "_ZN1A7SEi");

   function Constructor return A'Class;
   pragma CPP_Constructor (Constructor);
   pragma Import (CPP, Constructor, "_ZN1AC2Ev");
end My_Cpp_Interface;
```

With the above package we can now declare objects of type A and dispatch calls to the corresponding subprograms in the C++ side. We can also extend A with further fields and primitives, and override on the Ada side some of the C++ primitives of A.

It is important to note that we do not need to add any further information to indicate either the object layout, or the dispatch table entry associated with each dispatching operation. For completeness we have also indicated to the compiler that the default constructor of the object is also defined in the C++ side.

In order to further simplify interfacing with C++ we are currently working on a utility for GNAT that automatically generates the proper mangled name for the operations, as generated by the G++ compiler. This would make the pragma Import redundant.

7 Conclusion

We have described part of the work done by the GNAT Development Team to implement Ada 2005 interface types in a way that is fully compatible with the C++ Application Binary Interface (ABI). We have explained our implementation of abstract interface type conversions, including the special support required for variable sized tagged objects. We have also given an example that shows the power of the combined use of the GNAT and G++ compilers for mixed-language object-oriented programming.

The implementation described above is available to users of GNAT PRO, under a switch that controls the acceptability of language extensions (note that these extensions are not part of the current definition of the language, and can not be used by programs that intend to be strictly Ada95-conformant). This implementation is also available in the GNAT compiler that is distributed under the *GNAT Academic Program* (GAP) [15].

We hope that the early availability of the Ada 2005 features to the academic community will stimulate experimentation with the new language, and spread the use of Ada as a teaching and research vehicle. We encourage users to report their experiences with this early implementation of the new language, in advance of its much-anticipated official standard.

Acknowledgments

We wish to thank Robert Dewar, Cyrille Comar, Gary Dismukes, and Matthew Gingell for the discussions that helped us to clarify the main concepts described in this paper. We also wish to thank the dedicated and enthusiastic members of AdaCore, and the myriad supportive users of GNAT whose ideas, reports, and suggestions keep improving the system.

References

1. J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification (3rd edition)*. Addison-Wesley, 2005. ISBN: 0-321-24678-0.
2. E. International. *C# Language Specification (2nd edition)*. Standard ECMA-334. Standardizing Information and Communication Systems, December, 2002.
3. ISO/IEC. *Programming Languages: C++ (1st edition)*. ISO/IEC 14882:1998(E). 1998.
4. Ada Rapporteur Group. *Annotated Ada Reference Manual with Technical Corrigendum 1 and Amendment 1 (Draft 14): Language Standard and Libraries*. (Working Document on Ada 2005).
5. S. Taft, R. A. Duff, and R. L. Brukardt and E. Ploedereder (Eds). *Consolidated Ada Reference Manual with Technical Corrigendum 1. Language Standard and Libraries*. ISO/IEC 8652:1995(E). Springer Verlag, 2000. ISBN: 3-540-43038-5.
6. CodeSourcery, Compaq, EDG, HP, IBM, Intel, Red Hat, and SGI. *Itanium C++ Application Binary Interface (ABI)*, Revision 1.83, 2005. http://www.codesourcery.com.prev/cxx-abi
7. Ada Rapporteur Group. *Abstract Interfaces to Provide Multiple Inheritance*. Ada Issue 251.
   Available at http://www.ada-auth.org/cgi-bin/ cvsweb.cgi/AIs/AI-00251.TXT.
8. Ada Rapporteur Group. *Object.Operation Notation*. Ada Issue 252, Available at http://www.ada-auth.org/ cgi-bin/cvsweb.cgi/AIs/AI-00252.TXT.
9. Ada Rapporteur Group. *Protected and Task Interfaces*. Ada Issue 345, Available at http://www.ada-auth.org/cgi-bin/cvsweb.cgi/ AIs/AI-00345.TXT.
10. Ada Rapporteur Group. *Null Procedures*. Ada Issue 348, Available at http://www.ada-auth.org/cgi-bin/ cvsweb.cgi/AIs/AI-00348.TXT.
11. Ada Rapporteur Group. *Single Task and Protected Objects Implementing Interfaces*. Ada Issue 399. Available at http://www.ada-auth.org/cgi-bin/ cvsweb.cgi/AIs/AI-00399.TXT.
12. J. Miranda, E. Schonberg. *GNAT: On the Road to Ada 2005*. SigAda'2004, November 14-18, Pages 51-60. Atlanta, Georgia, U.S.A.
13. J. Miranda, E. Schonberg, G. Dismukes. *The Implementation of Ada 2005 Interface Types in the GNAT Compiler*. 10th International Conference on Reliable Software Technologies, Ada-Europe'2005, 20-24 June, York, UK.
14. J. Miranda, E. Schonberg, K. Kirtchov. *The Implementation of Ada 2005 Synchronized Interfaces in the GNAT Compiler*. SigAda'2005, November 13-17. Atlanta, Georgia, U.S.A.
15. AdaCore. *GNAT Academic Program*. http://www.adacore.com/academic_overview.php