Multi-Language Programming: The Challenge and Promise of Class-Level Interfacing

Cyrille Comar, Matthew Gingell, Olivier Hainque, Javier Miranda
AdaCore
{comar, gingell, hainque, miranda}@adacore.com

Abstract

Many computer applications today involve modules written in different programming languages, and integrating these modules together is a delicate operation. This first requires the availability of formalisms to let programmers denote "foreign" entities like objects and subprograms as well as their associated types. Then, proper translation of what programmers express often calls for significant implementation effort, possibly down to the specification of very precise ABIs (Application Binary Interfaces). Meta-language based approaches a-la CORBA/IDL are very powerful in this respect but typically aim at addressing distributed systems issues as well, hence entail support infrastructure that not every target environment needs or can afford. When component distribution over a network is not a concern, straight interfacing at the binary object level is much more efficient. It however relies on numerous low level details and in practice is most often only possible for a limited set of constructs.

Binary level interaction between foreign modules is traditionally achieved through subprogram calls, exchanging simple data types and relying on the target environment's core ABI. Object Oriented features in modern languages motivate specific additional capabilities in this area, such as class-level interfacing to allow reuse and extension of class hierarchies across languages with minimal constraints. This paper describes work we have conducted in this context, allowing direct binding of Ada extensible tagged types with C++ classes. Motivated by extensions to the Ada typing system made as part of the very recent language standard revision, this work leverages the GCC multi-language infrastructure and implementation of the Itanium C++ ABI. We will first survey the issues and mechanisms related to basic inter-language operations, then present the interfacing challenges posed by modern object oriented features after a brief overview of the Ada, C++, and Java object models. We will continue with a description of our work on Ada/C++ class-level interfacing facilities, illustrated by an example.

1 Interfacing Across Programming Languages - Introduction

Two general aspects of Multi-Language Programming are the formalisms available to denote and use "foreign" entities exposed from a different language than the one in which they are referred to, and the support infrastructure for what programmers express. "Interfacing" can cover many different things, such as access to foreign data, foreign type representa-

tion, calls to foreign subprograms, handling of foreign events like exceptions, and reuse of object class hierarchies. In any case, an interface always implies agreement between the involved parties. For instance, a subroutine call will only operate properly if the caller and the callee agree on how arguments are passed (in what order, using what machine resources), who allocates/releases this or that part of the stack, how aligned the stack pointer is expected to be, etc. Likewise, operating on a foreign variable requires a way to describe or denote the variable's "native" type to ensure a correct interpretation of the actual value layout. Typically, more powerful formalisms make programmers lives easier at the price of more complicated underlying infrastructure.

1.1 Core Mechanisms - Basic Capabilities

A first set of basic interfacing possibilities is provided by explicit programming language features associated with well established *calling conventions* and low level rules for the target environment specified in base *Application Binary Interface* (ABI) documents.

Among other things, base ABI documents describe binary files formats, basic data type layouts, stack frame organization, and machine level conventions for passing parameters to and returning results from subprograms. See [19] and [12] for examples of such documents for the i386 and amd64 architectures. Additional calling conventions may apply in some environments, such as the `stdcall`/`fastcall` variants on x86-Windows [17], or for some specific programming languages as illustrated by the differences between Pascal and C in arguments passing order. These conventions provide a common ground for basic inter-language interfacing capabilities and binary code interoperability, ensuring for instance proper interaction between GCC compiled code and target operating system libraries.

On top of the common base conventions we have just surveyed, various standard devices are available on the programming languages side. As a first example, the Ada Reference Manual (ARM) includes a full annex dedicated to the issue [22, Annex B], covering interfacing with C, Cobol, and Fortran, and allowing implementations to support other languages. The minimum support specified in this annex consists of standard packages for each language, for instance the `Interfaces.C` hierarchy for C, and specific compiler *pragmas* :

- *Pragma Import*, to import an entity defined in a foreign language into an Ada program, thus allowing a foreign-language subprogram to be called from Ada, or a foreign-language variable to be accessed from Ada.
- *Pragma Export*, to export an Ada entity to a foreign language.
- *Pragma Convention*, to specify that an Ada entity should use the conventions of another language for passing parameters to subprograms, or else to represent a data type in memory (for example determining matrix element ordering).
- *Pragma Linker_Options*, to specify the system linker parameters needed when a given compilation unit is included in a program.

The following code example illustrates the use of some of these facilities to call a C function from Ada to print out an `int` value found at a provided address. It uses the standard `Interfaces.C` package to get access to the Ada type corresponding to `int`, declares an Ada subprogram to represent the C service interface, and imports the service by way of an

`Import` pragma. The latter tells the compiler that the subprogram is external with C convention and states what symbol (link name) should be used to refer to it.

```
with Interfaces.C; use Interfaces;
procedure Binding_Example is

   -- Map and use C function
   -- void dump_int_at (int *ptr);

   procedure Dump_Int_At (Ptr : access C.Int);

   pragma Import
    (Convention => C,
     Entity     => Dump_Int_At,
     Link_Name  => "dump_int_at");

   Myint : aliased C.Int := 12;
begin
   Dump_Int_At (Myint'Access);
end;
```

The Ada `Access` *attribute* used here in `Myint'Access` corresponds to the `&` unary addressing operator in C: It produces an address, called an *access value*, said to designate the entity. Ada access values are normally subject to *accessibility checks* mandated by the language to prevent the creation of dangling pointers [22, 3.10.2-24]. Roughly, an access value may only be assigned to an object of an access type if the value lifetime is guaranteed to be shorter than the lifetime of the target type. Performing these checks requires run-time code in some cases, raising the predefined *Program_Error* exception in case of failure. With GNAT, accessibility checks result in automatic extra argument passing in calls to subprograms with access parameters. A noticeable effect of the C convention applied to `Dump_Int_At` in our example is to disable this circuitry, as the extra parameter is not part of the base interface and only makes sense for Ada subprograms.

As other examples, C++ provides *linkage specifications* such as `extern "C"` to allow the use of C++ entities in other languages, and calls to foreign routines from Java are possible thanks to an exhaustive Java Native Interface specification [11].

1.2 Higher Level Facilities and Paper Overview

As time goes by, programming languages evolve, higher level features are introduced, implementation choices are made, and binary compatibility issues, especially with respect to other languages, are not always part of the picture upfront. This is legitimate, as a concept in one language doesn't necessarily have a counterpart in others, and because complex factors come into play views inevitably vary on what scheme is best in each specific case. Still, commonalities do occur even for sophisticated features, and the capability to interface across languages at these higher levels is often desirable and an interesting challenge. For instance, an Ada top-level subprogram might be interested in catching exceptions raised by C++ subcomponents, or vice-versa. Although the concept of "exception" is similar in both languages, there are variations in the way it is precisely mapped on each side, and determining the appropriate semantics for such a facility is difficult to start with.

This paper describes work we have conducted in this context, on GNU Ada/C++ "class-level interfacing," to allow direct binding of Ada extensible tagged types hierarchies to C++ classes in both directions. Motivated by extensions to the Ada typing system made as part of the very recent language standard revision [1], this work leverages the GCC multi-language infrastructure and implementation of the Itanium C++ ABI [5] to simplify interfacing between OO languages at the class-level.

In Section 2 we briefly describe the Ada OO model and its relationship with the C++ and Java models. In Section 3 we present in greater detail what "class-level interfacing" involves and the various possible approaches. In Section 4 we analyze the GNAT specific capabilities for interfacing Ada with C++, illustrated

with a commented example in Section 5, and then offer our conclusions.

2 Static OO Models Comparison: Ada, Java, C++

Booch [2] defines Object Orientation around seven principles: Abstraction, Encapsulation, Modularity, Hierarchy, Typing, Concurrency and Persistence. The first two principles are about separating how objects are defined and used from how they are represented and implemented. Modularity is about organizing programs as a collection of separate components with defined interactions and limited access to data. Typing and Hierarchy are about distinguishing different kinds of objects and structuring them according to their common characteristics. A complete description of these principles can also be found in [9, Section 1.3.2].

In C++ and Java, the notion of "class" is central to all these principles even though modularity is also achieved through name-spaces and separate files. In those languages, classes allow grouping of data members along with their associated function members (methods). They also specify their position in a hierarchy by specifying their immediate parents and offer visibility restriction mechanisms for their members.

In Ada, the first three concepts (Abstraction, Encapsulation, and Modularity) are associated with packages and "private" declarations while the Typing concept is clearly associated with the Ada typing model. The Hierarchy principle is found both in packages and types: the child package construct allows the programmer to define a hierarchy of packages, and the type derivation allows him to create hierarchies of types.

Ada 83, Ada's original definition, was considered an Object-Based language. It was based on the above principles without offering any mechanism for dynamic polymorphism. In fact, dynamic dispatching was deliberately banned from the language since it was, at the time, considered incompatible with its safety requirements. In this first model, a class is represented by a private type along with its primitive operations (methods) encapsulated in a package. The implementation of the private type is typically a record grouping all data members, and the implementation of methods are hidden in the package body.

The second revision of the language, known as Ada 95, enriches its typing system with a new variety of record called "tagged" records. The main characteristic of these records is that they can be extended during derivation and thus are used as the basis for dynamic polymorphism under a single inheritance model. Both C++ and Java fully support single inheritance. Contrary to those languages where polymorphism is implicit, Ada distinguishes it through an explicit notation: T'Class is the polymorphic, called *class-wide*, version of a specific tagged type T, which means that the actual run-time type of an object declared of type T'Class can be T or any of its descendant.

In Java, all methods are dispatching. In C++, methods are dispatching when they are declared "virtual". In Ada, all methods are potentially dispatching and a call dispatches or not depending on the nature of the object it applies to. Dispatching will only occur when the latter has a polymorphic type, as illustrated by the code excerpt below:

```
-- A call is dispatching if the controlling
-- argument type is classwide:

X : T'Class := ...;
Y : T       := ...;
...
X.T_Method;      -- dispatching
Y.T_Method;      -- not dispatching
```

C++ offers full-scale multiple inheritance. That is to say, a class may have several parents and inherits all their data and function members. This is a powerful capability providing a great deal of expressive power. At the design level, it is particularly convenient for composing concepts represented by independent classes. Programming with full multiple inheritance requires familiarity with the answers provided by the language to tricky questions such as: What happens when a class inherits multiple times from the same ancestor through different derivation paths? What happens when inheriting methods with the same profile from different parents? A thorough overview of how C++ answers such questions is available from [21], along with many ideas on how multiple inheritance can be implemented efficiently. Nonetheless, although multiple inheritance has proven to be a very powerful paradigm for skilled programmers, its extensive use may have negative consequences for the readability and long term maintainability of software.

In recent years, a number of language designs [6, 7] have adopted a compromise between full multiple inheritance and strict single inheritance, which is to allow multiple inheritance of *specifications*, and 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. This model has much of the power of full-scale multiple inheritance, but without most of the implementation and semantic difficulties of the C++ multiple inheritance model [10].

Ada 2005 provides support for such abstract interface types [1, Section 3.9.4]. Its characteristics are introduced by means of an interface type declaration and a set of subprogram declarations. The interface type has no data components and its primitive operations are either abstract or null, in which case they behave as if their body was empty. A data type that implements an interface must provide non-abstract versions of all the abstract operations of its parents. Here is a code sample to illustrate the declaration of interface types and the associated multiple inheritance capability in Ada 2005:

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

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

  type Root is tagged record with private;
  procedure A (Obj : T);
  function B (Obj : T) return Integer;

  type DT is
    new Root and I1 and I2 with private;
  -- DT1 must implement P, and R
  ...

  type DT2 is new DT with private;
  -- Inherits all the primitives and
  -- interfaces of the ancestor

private
  type Root is tagged record with
    -- Root components
    ...
  end record;

  type DT is
    new Root and I1 and I2 with record
    -- DT components
    ...
  end record;

  type DT2 is new DT with record
    -- DT2 components
    ...
  end record;
end Interfaces_Example;
```

The interface I1 has one subprogram, *P*. The interface I2 has the same operations as I1 plus two subprograms: the null subprogram *Q* and the abstract subprogram *R*. Then, we define the root of a derivation class that has two primitive operations, *A* and *B*. *DT* extends the root type and also inherits the two interfaces I1 and I2, so it is required to implement all the associated abstract subprograms. Finally, type *DT2* extends

DT1, inheriting all the primitive operations and interfaces of its ancestor.

OO languages that provide abstract interface types [6, 7] have a run-time mechanism that determines 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 consider 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
  -- 1: dispatch call
  ... := P (Obj);

  --    2: membership test
  if Obj in I2'Class then

    -- 3: interface conversion plus
    --      dispatch call
    R (I2'Class (Obj));
  end if;

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

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

Ada 2005 also extends abstract interfaces for its use in concurrency, but this topic is not discussed in this paper. For details on the GNAT implementation of synchronized interfaces see [15].

3 Interfacing at the class level

3.1 Basic Requirements

In general, reusing an object-oriented system requires two distinct capabilities: creating instances of existing classes and defining new classes inheriting from them. Reusing an OO system written in a different language requires the additional capability: to "see" foreign classes and use them with as few restrictions as possible. In particular it implies the possibility of defining in one language an instance of a class which has been implemented in another. Another interesting capability is inheriting from foreign classes, which implies that dynamic binding can cross language boundaries transparently. Although of less general interest, Run Time Type Information (RTTI) queries such as membership tests are also worth mentioning.

For such interfacing capabilities to make sense, minimal commonalities between the OO models are required to preserve coherence between a class hierarchy defined on one side and used on the other.

3.2 Common Approaches

A well-known approach to inter-language class-level interfacing consists of resorting to a common meta language. CORBA [18] offers an interesting case of the definition of such a model. CORBA's main goal is to support the development of Object-Oriented distributed systems. Thus inter-computer communication

plays an important role. If we abstract the communication component however, CORBA offers a model for interfacing systems that may be written in different languages and thus offers a language independent object model. This model is described using an Interface Definition Language (IDL). The CORBA IDL defines the concepts needed to describe the most common abstractions: basic and composite data types, modules, exceptions, and class hierarchies, possibly with multiple inheritance. Not being an implementation language, only the definition part of a class needs to be provided in the CORBA IDL and corresponds to a Java interface. In fact they are also called interfaces in IDL jargon. Hence, CORBA IDL seems an ideal solution for interfacing at the class level since it offers the common ground on which languages with different object models can usefully communicate.

At the practical level, however, the situation is not ideal. Within the CORBA framework, each language requires a binding between its native OO model and the Definition Language, an IDL compiler is needed to transform IDL models into a set of native specifications or header files, and these then have to be connected to the existing system. So, not only does the user need to learn and use a yet another language, the final system ends up with a thick layer for the interfacing part composed of the two bindings mentioned above connected by a complete communication middleware (Object Request Broker, ORB). In situations where the various subsystems are not intended to be deployed on different machines, this can represent a very significant overhead both in development effort and in the amount of code dedicated to interfacing.

The use of an Interface Definition Language is not limited to CORBA. It is also used in other contexts where interfacing at class level is sought. [4] offers a good description of such a case for interfacing two languages with quite different OO models: OCaml and C#. The IDL used in this context is very close to Java syntax and the paper gives a good description of the notion of shadow (or Proxy) classes, another typical model for class level interfacing between two incompatible worlds.

The "shadow/proxy class" idea is to define two matching class hierarchies on each side of the language fence. For each class implemented on one side, a shadow class is defined on the other side where all its methods are wrappers that ultimately call the corresponding foreign method. On the shadow side, each class instance needs to be associated to a real instance on the other side, which can be done as part of the initialization of the shadow instance.

The SWIG system [20] is worth mentioning in this context. SWIG is a software development tool that connects programs written in C and C++ with a variety of high-level programming languages such as Java, Python, Ruby or Scheme, most of which offer their own OO model. As with CORBA, SWIG uses an IDL. Its syntax is very close to C/C++ header files, so interface files can be written quickly by simplifying the existing header files of the system to interface. SWIG automatically creates the hierarchy of shadow classes that will allow those various OO languages to access pre-existing C++ class hierarchies.

The shadow class mechanism becomes complicated when the original language features garbage collection, since the shadow object may end up being the only valid reference to the real object and is usually hidden from the original environment. When the language does not provide garbage collection, the opposite problem can arise: how to make sure that those shadow objects or their counterpart are released properly before becoming unreachable? All these issues are described in great detail in the SWIG documentation.

Apart from the aforementioned families of approaches, direct interfacing at the binary level can sometimes be achieved, alleviating the need for intermediate software layers. This is what we have done for Ada/C++ interfacing with the GNAT compiler, as described in the following section.

4 The GNAT Approach to Ada/C++ Interfacing

The interfacing mechanisms mentioned in the previous section have been designed to be independent of compiler technologies. They generate potentially heavy glue code whose only requirements are related to the semantics of the languages to interface and not to their actual implementation.

As compiler implementors with full control over code generation on one side of the interface, our perspective is different. Our purpose is to provide a low-level mechanism that simplifies interfacing and allow production of lighter glue code when possible. For instance, when an object is part of an Ada/C++ interface, a heavy duty interfacing mechanism such as CORBA requires the following steps: 1) marshall the object to transform it from its Ada representation to a machine independent representation such as CDR in the CORBA case; 2) send this encoded data through the communication channel (ORB for CORBA); and 3) unmarshall the data into its C++ representation.

From the compiler viewpoint, a much simpler method can be used if one side can mimic the data representation expected by the other. In such a situation interfacing becomes as simple as sharing a name or a reference. In this context, our goal is to extend the base Ada interfacing pragmas introduced in Section 1.1 to encompass the class concept and its associated mechanisms, such as dynamic dispatching. This is possible thanks to the commonalities between the Ada and the C++ object models: a C++ class maps naturally to an Ada tagged type, a class data member is a tagged record component, a virtual function member maps to an Ada primitive operation, and static members functions or constructors can be mapped to Ada operations on the classwide type. The following subsections describe two different schemes we have developed to achieve this goal.

4.1 Original Scheme for Ada95

When the original Ada95/C++ interfacing mechanism was designed in the mid 90s, a study of various C++ compilers showed wide variation in the layout of C++ objects and their virtual function tables. As a consequence, we decided to provide a model of interfacing to C++ which depended as little as possible on the choices made by particular C++ implementations. In this approach, the GNAT compiler made no assumptions about how objects generated by the C++ compiler were laid out, and required that the user determine and provide a correct matching representation in Ada themselves.

For instance, the compiler made no assumptions about where a virtual function table pointer would appear in an imported object. Hence, in the declaration of the corresponding type in Ada the user had to provide a dummy pointer field and mark it explicitly with a `pragma CPP_Vtable`. Additionally the compiler had no special knowledge of how a virtual function table was actually laid out, leaving it up to the user to determine whether or not he needed to provide specific offsets in his method bindings via `pragma CPP_Virtual`.

In addition, no knowledge about what might be needed to call a C++ method was encapsulated in the compiler itself. Instead, the compiler delegated the responsibility for accessing the vtable and calling methods through it to a set of routines in the run-time with a well defined procedural interface. This abstraction meant it was possible to adapt GNAT to changes in C++ compilers or to adapt it to new compilers very easily at the run-time level without actually having to make any changes in the compiler itself.

On the one hand, this approach enabled a sufficiently motivated user to find a way of interfacing to C++ objects generated by a wide variety of compilers. For instance, users interested enough in finding the virtual function pointer in objects generated by the Sun C++ compiler and determining at what offsets it had placed what methods could, with enough effort, put together a useful Ada binding.

On the other hand, this process was labor intensive and error prone, and required a level of knowledge about the implementation of both compilers that the user may not have had and was unlikely to be interested in acquiring. While in principle the facilities the user required were provided, in practice there was a great deal left to be desired.

4.2 Redesign for Ada 2005 - Leveraging the C++ ABI

An alternate approach recently added to the GNAT compiler takes advantage of knowledge of the C++ ABI [5]. This approach takes responsibility for the details and complexities which the previous approach left to the end user. This ABI is also followed by GCJ, the GNU Java compiler [8, Section 12.1]. For each tagged type the compiler generates a primary dispatch table associated with its single-inheritance line of derivation and a secondary dispatch table for each abstract interface type inherited by the tagged type. This model incurs storage costs, in the form of additional pointers to dispatch tables in each object and *thunks* that adjust the value of the pointer to the object implementing abstract interface types.

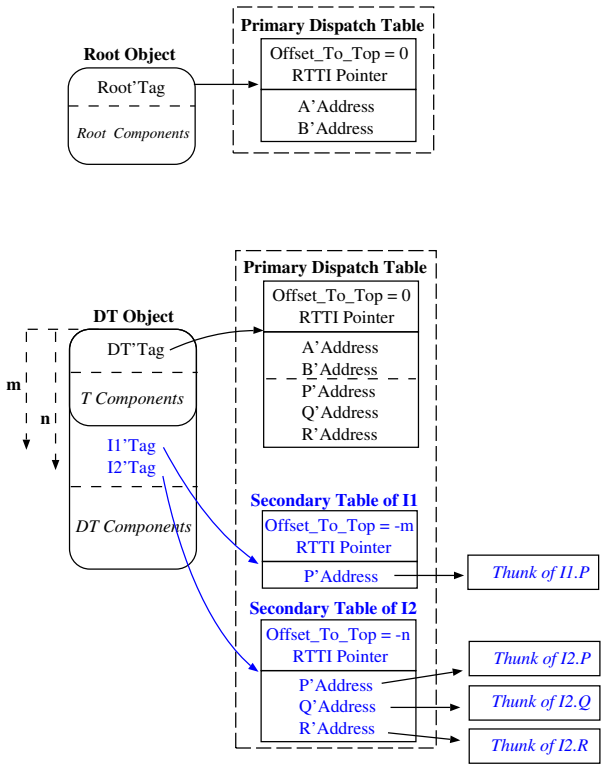

Figure 1: Layout compatibility with C++

Following with the example presented in section 2, Figure 1 represents the layout of the tagged types *Root* and *DT*. 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. 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 DT, type derived from Root that implements two interfaces (I1 and I2). The layout of the object (left side of the figure), shows 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. The offset-to-top values of interfaces I1 and I2 are *m* and *n* respectively. This offset provides a way to find the top of the object from any derived object that contains secondary dispatch tables and is necessary in type conversions. 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.

The main difference between the current ABI layout provided by the Ada compiler and the official C++ ABI [5] is the contents of the RTTI pointer. On the Ada side this pointer references a record containing information required to support Ada semantics (accessibility level, expanded name of the tagged type, etc.) plus two additional tables: a table containing the tag of all the immediate ancestors of the type, and a table containing the tag of all the abstract interface types implemented by the type plus its corresponding offset-to-top values in the object layout. These tables give run-time support to the membership test and interface conversions respectively. Figure 2 completes the run-time data structure described in previous section with the GNAT *Type Specific Data* record.

It is clear that this difference introduces several incompatibilities. For example, on the Ada side we cannot make use of the membership test on a class imported from the C++ side, and similarly on the C++ side the dynamic cast operator cannot be used with tagged types imported from the Ada side. We are working on this area to reduce these layout differences.

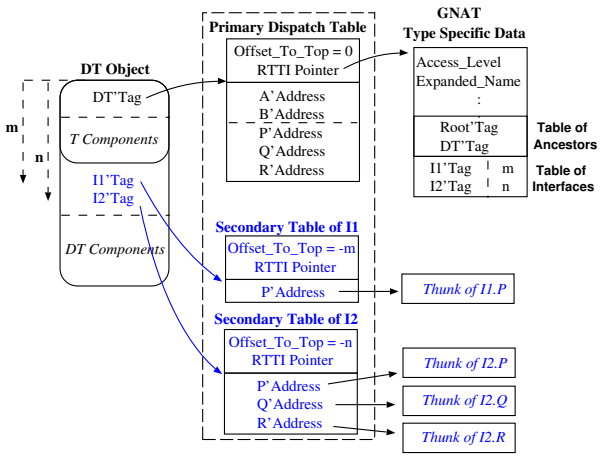

Figure 2: GNAT Layout

Regarding the C++ ABI's [5] completeness for use in the implementation of other OO languages, we have found that the case of variable sized tagged objects is not supported. Complications arise when a tagged type has a parent that includes some component whose size is determined by a discriminant and the type is also derived from abstract interface types. 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 *DT* 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 because these tables are shared by all the objects of the type. The C++ ABI does not address this problem for the simple reason

that C++ classes do not have non-static components.

In order to solve this problem we decided 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 reach the object's base address, but we also 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 generates object specific functions which read the value of the offset-to-top hidden field, and stores pointers to these functions in the table of interfaces. For further information see [16].

5 A Commented Example

In this section we present the new GNAT features for interfacing with C++ by means of an example. This example consists of a classification of animals; classes have been used to model our main classification of animals, and interfaces provide support for the management of secondary classifications. We will first present a case in which the types and constructors are defined on the C++ side and imported from the Ada side, and latter the reverse case.

5.1 Importing from C++

The root of our derivation will be the *Animal* class, with a single private attribute (the *Age* of the animal) and two public primitives to set and get the value of this attribute.

```
class Animal {
 public:
   virtual void Set_Age (int New_Age);
   virtual int Age ();
 private:
   int Age_Count;
};
```

Abstract interface types are defined in C++ by means of classes with pure virtual functions and no data members. In our example we will use two interfaces that provide support for the common management of *Carnivore* and *Domestic* animals:

```
class Carnivore {
public:
   virtual int Number_Of_Teeth () = 0;
};

class Domestic {
public:
   virtual void Set_Owner (char* Name) = 0;
};
```

Using these declarations, we can now say that a *Dog* is an animal that is both Carnivore and Domestic, that is:

```
class Dog : Animal, Carnivore, Domestic {
 public:
   virtual int  Number_Of_Teeth ();
   virtual void Set_Owner (char* Name);

   Dog(); // Constructor
 private:
   int  Tooth_Count;
   char *Owner;
};
```

In the following examples we will assume that the previous declarations are located in a file named *animals.h*. The following package demonstrates how to import these C++ declarations from the Ada side:

```
with Interfaces.C.Strings;
use  Interfaces.C.Strings;
package Animals is
  type Carnivore is interface;
  function Number_Of_Teeth (X : Carnivore)
    return Integer is abstract;
  pragma Convention (CPP, Number_Of_Teeth);

  type Domestic is interface;
  procedure Set_Owner
    (X    : in out Domestic;
     Name : Chars_Ptr) is abstract;
  pragma Convention (CPP, Set_Owner);

  type Animal is tagged private;
  pragma CPP_Class (Animal);

  procedure Set_Age
    (X : in out Animal; Age : Integer);
  pragma Import (CPP, Set_Age);

  function Age (X : Animal) return Integer;
  pragma Import (CPP, Age);

  type Dog is new Animal
    and Carnivore and Domestic with private;
  pragma CPP_Class (Dog);

  function Number_Of_Teeth (A : Dog)
    return Integer;
  pragma Import (CPP, Number_Of_Teeth);

  procedure Set_Owner
    (A : in out Dog; Name : Chars_Ptr);
  pragma Import (CPP, Set_Owner);

  function New_Dog return Dog'Class;
  pragma CPP_Constructor (New_Dog);
  pragma Import (CPP, New_Dog, "_ZN3DogC2Ev");
private
  type Animal is tagged record
    Age : Integer := 0;
  end record;

  type Dog is new Animal
    and Carnivore and Domestic with
  record
    Tooth_Count : Integer;
    Owner       : Chars_Ptr;
  end record;
end Animals;
```

Thanks to the compatibility between GNAT run-time structures and the C++ ABI, interfacing with these C++ classes is easy. The only requirement is that all the primitives and components must be declared exactly in the same order in the two languages. The code makes no use of the GNAT specific pragmas CPP_Vtable and CPP_Virtual described in Section 4.1.

Regarding the abstract interfaces, we must indicate to the GNAT compiler by means of a `pragma Convention (CPP)`, the convention used to pass the arguments to the called primitives will be the same as for C++. For the imported classes we use `pragma CPP_Class` to indicate that they have been defined on the C++ side; this is required because the dispatch table associated with these tagged types will be built on the C++ side and therefore will not contain the predefined Ada primitives which Ada would otherwise expect.

Finally, for each user-defined primitive operation we must indicate by means of a `pragma Import (CPP)` that they are imported from the C++ side.

As the reader can see there is no need to indicate the C++ mangled names associated with each subprogram because it is assumed that all the calls to these primitives will be dispatching calls. The only exception is the constructor, which must be registered in the compiler by means of `pragma CPP_Constructor` and needs to provide its associated C++ mangled name because the Ada compiler generates direct calls to it. In order to further simplify interfacing with C++, we are currently working on a utility for GNAT that automatically generates the proper mangled names for C++ imported subprograms, as generated by the G++ compiler.

With the above packages we can now declare objects of type Dog on the Ada side and dispatch calls to the corresponding subprograms on the C++ side. We can also extend the tagged type Dog with further fields and primitives, and override some of its C++ primitives on the Ada side. For example, here we have a type derivation defined on the Ada side that inherits all the dispatching primitives of the ancestor from the C++ side.

```
with Animals; use Animals;
package Vaccinated_Animals is
   type Vaccinated_Dog is
      new Dog with null record;
   function Vaccination_Expired
     (A : Vaccinated_Dog) return Boolean;
   pragma Convention
     (CPP, Vaccination_Expired);
end Vaccinated_Animals;
```

It is important to note that, because of the ABI compatibility, the programmer does not need to add any further information to indicate either the object layout or the dispatch table entry associated with each dispatching operation.

5.2 Exporting to C++

Now let us define all the types and constructors on the Ada side and export them to C++, using the same hierarchy of our previous example:

```
with Interfaces.C.Strings;
use Interfaces.C.Strings;
package Animals is
   type Carnivore is interface;
   function Number_Of_Teeth (X : Carnivore)
     return Integer is abstract;
   pragma Convention (CPP, Number_Of_Teeth);

   type Domestic is interface;
   procedure Set_Owner
     (X     : in out Domestic;
      Name : Chars_Ptr) is abstract;
   pragma Convention (CPP, Set_Owner);

   type Animal is tagged private;
   pragma Convention (CPP, Animal);

   procedure Set_Age
     (X    : in out Animal;
      Age : Integer);
   pragma Export (CPP, Set_Age);

   function Age (X : Animal) return Integer;
   pragma Export (CPP, Age);

   type Dog is new Animal
     and Carnivore
     and Domestic with private;
   pragma Convention (CPP, Dog);

   function Number_Of_Teeth (A : Dog)
     return Integer;
   pragma Export (CPP, Number_Of_Teeth);

   procedure Set_Owner
     (A     : in out Dog;
      Name : Chars_Ptr);
   pragma Export (CPP, Set_Owner);

   function New_Dog return access Dog'Class;
   pragma Export (CPP, New_Dog);

private
   type Animal is tagged record
     Age : Integer := 0;
   end record;

   type Dog is new Animal
     and Carnivore and Domestic with
   record
     Tooth_Count : Integer;
     Owner       : Chars_Ptr;
   end record;
end Animals;
```

Compared with our previous example the only difference is the use of pragma *Export* to indicate to the GNAT compiler that the primitives will be available to C++. Thanks to the ABI compatibility, on the C++ side there is nothing else to be done; as explained above, the only requirement is that all the primitives and components are declared in exactly the same order. For completeness, let us see a brief C++ main program that uses the declarations available in *animals.h* (presented in our first example) to import and use the declarations from the Ada side, properly initializing and finalizing the Ada run-time system along the way:

```
#include "animals.h"
#include <iostream>
using namespace std;

void Check_Carnivore (Carnivore *obj) { ... }
void Check_Domestic (Domestic *obj)   { ... }
void Check_Animal (Animal *obj)       { ... }
void Check_Dog (Dog *obj)             { ... }

extern "C" {
  void adainit (void);
  void adafinal (void);
  Dog* new_dog ();
}

void test ()
{ Dog *obj = new_dog();  // Ada constructor
  Check_Carnivore (obj); // Check secondary DT
  Check_Domestic (obj);  // Check secondary DT
  Check_Animal (obj);    // Check primary DT
  Check_Dog (obj);       // Check primary DT
}

int main ()
{
  adainit (); test(); adafinal ();
  return 0;
}
```

6 Conclusion

The C++ ABI [5] was first defined as part of a new processor ABI, but it has evolved into a processor independent ABI for C++ which can be used as a de-facto standard for other languages (ie. currently the GNU C++, Ada and Java compilers support this ABI). This evolution not only allows mixing C++ objects compiled with different compilers in the same executable, but also allows multi-language object-oriented programs compiled into a single executable. The common ABI allows the programmer to mix objects from different languages and also permits him the use of features such as dynamic dispatching, which are not limited by language boundaries.

It is well known that several modern static Object-Oriented languages offer similar support for single inheritance and multiple inheritance of abstract interface types. However, the current C++ ABI does not completely fulfill all the requirements of these languages. For example, in this paper we have shown that this ABI should be extended for languages with variable sized objects like Ada. We think that it would be desirable to extend this ABI with new sections covering the basic data structures supporting Object Oriented features, such as dynamic dispatching, in a language independent way to give GCC full support to multi-language programming at the class level. This would improve interfacing capabilities between the OO languages supported by GCC and would open new opportunities for software reuse in a world where programming language trends evolve rapidly.

For this work to be of direct use to the GCC users interested in reusing libraries written in several languages (ie. Ada, C++, Java), a tool for automating the generation of the interface files would be hightly desirable. SWIG [20] seems to offer a very promising framework for developing such a tool since it provides all the technology for generating shadow class hierarchies. In this context, languages with ABI compatibility have an important benefit: shadow methods would not be wrappers anymore but "direct" views of the real methods because the need for shadow objects can be replaced by direct views of the real object, thus improving the efficiency of the code and eliminating all the complexity related to memory management.

References

[1] Ada Rapporteur Group. *Annotated Ada Reference Manual with Technical Corrigendum 1 and Amendment 1 (Draft 16): Language Standard and Libraries*. (Working Document on Ada 2005). Ada-Europe, 2006.

[2] G. Booch *Object-Oriented Analysis and Design* Addison-Wesley, 2nd edition,1993. ISBN: 0805353402

[3] J. Byous. *Java Technology: The Early Years*, 2006. `http://java.sun.com/features/1998/05/birthday.html`.

[4] E. Chailloux, G. Grégoire, R. Montelatici *Mixing the Objective Caml and C+ Programming Models in the .NET Framework* The 3rd International Conference on .NET Technologies, Plenz, Czech Republic May 30-June 1, 2005

[5] CodeSourcery, Compaq, EDG, HP, IBM, Intel, Red Hat, and SGI. *Itanium C++ Application Binary Interface (ABI)*, Revision 1.86, 2005. `http://www.codesourcery.com/cxx-abi`

[6] E. International. *C# Language Specification (2nd edition)*. Standard ECMA-334. Standardizing Information and Communication Systems, December, 2002.

[7] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification (3rd edition)*. Addison-Wesley, 2005. ISBN: 0-321-24678-0.

[8] *Guide to GNU GCJ*, 2005. `http://gcc.gnu.org/onlinedocs/gcj/`

[9] *Handbook for Object-Oriented Technology in Aviation* `http://www.faa.gov/aircraft/air_cert/design_approvals/air_software/oot/`

[10] ISO/IEC. *Programming Languages: C++ (1st edition)*. ISO/IEC 14882: 1998(E). 1998.

[11] S. Liang. *The Java Native Interface: Programmer's Guide and Specification*. ISBN: 0-201-32577-2, Addison-Wesley Professional; 1st edition (June 10, 1999).

[12] M. Matz, J. Hubicka, A. Jaeger, M. Mitchell. *System V Application Binary Interface - AMD64 Architecture Processor Supplement*. June 2005, Available from `http://www.x86-64.org/documentation/`

[13] J. Miranda, E. Schonberg. *GNAT: On the Road to Ada 2005*. SigAda'2004, November 14-18, Pages 51-60. Atlanta, Georgia, U.S.A.

[14] 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.

[15] 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.

[16] J. Miranda, E. Schonberg. *Abstract Interface Types in GNAT: Conversions, Discriminants, and C++*. 11th International Conference on Reliable Software Technologies, Ada-Europe'2006, June, Porto, Portugal.

[17] N. Trifunovic. *Calling Conventions Demystified*. `http://www.codeproject.com/cpp/calling_conventions_demystified.asp`

[18] Object Management Group. *Common Object Request Broker Architecture: Core Specification* Version 3.0.3, March 2004.

[19] *System V Application Binary Interface - Intel 386 Architecture Processor Supplement*. Prentice Hall Trade, Third Edition 1994, ISBN: 0-131-04670-5. Fourth Edition available from `http://www.caldera.com/developers/devspecs/abi386-4.pdf`

[20] *Welcome to SWIG*. `http://www.swig.org/`

[21] B. Stroustrup *Multiple Inheritance for C++* The C/C++ Users Journal, May 1999 issue `http://www-plan.cs.colorado.edu/diwan/class-papers/mi.pdf`

[22] 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.