The Implementation of Ada 2005 Synchronized Interfaces in the GNAT Compiler

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

Edmond Schonberg
Courant Institute of Mathematical Sciences
New York University
U.S.A.
and AdaCore
schonberg@cs.nyu.edu

Hristian Kirtchev
AdaCore
U.S.A.
kirtchev@adacore.com

ABSTRACT

One of the most important object-oriented features of the new revision of the Ada Programming Language is the introduction of Abstract Interfaces to provide a form of multiple inheritance. Ada 2005 Abstract Interface Types are akin to Java interfaces, and as such support inheritance of specification rather than inheritance of implementation. Ada 2005 interfaces apply as well to tasks and protected types, and provide a classification mechanism for concurrent programming that goes considerably beyond the capabilities of Java.

This paper summarizes the implementation in the GNAT compiler of the various kinds of interfaces that relate to concurrent programming in Ada 2005 [1]. The implementation is efficient, and involves mostly modifications to the compiler front-end, with virtually minimal impact on run-time structures, beyond those that are in place to support regular interfaces. However, the implementation of interface operations as triggers in selective waits and asynchronous transfers of control proved to be surprisingly delicate and requires additional predefined primitive operations.

Categories and Subject Descriptors: D.3.2 [Programming Languages]: Language Classifications —Ada, D.3.3 [Language Constructs and Features]: classes and objects, inheritance, polymorphism, abstract data types, concurrent programming structures.

General Terms: Languages, Standardization, Reliability.

Keywords: Ada 2005, Interfaces, Synchronization, GNAT, Compiler.

1. INTRODUCTION

During the design of Ada 95 [2] there was much debate about whether the language should incorporate multiple inheritance. The outcome of the debate was to support single inheritance only. In recent years, a number of language designs [3, 4] 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: an interface type has no data components and no operation implementations. A type may implement multiple interfaces (its *progenitors*), but can inherit code for primitive operations from only one parent type [1, 6]. 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++ [5].

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 can be reused). Interface inheritance and dynamic dispatching on interface operation requires additional data structures that extend the single dispatch table model of Ada 95. The GNAT compiler implements interface inheritance by means of secondary dispatch tables (see citemsd05 for details). This model was chosen for its time efficiency, and its compatibility with the run-time structures used by G++, in order to simplify mixed-language object-oriented programming.

However, protected and task types that implement interfaces require further work: extra code has to be generated to support dispatching calls to protected and task entries as well as protected subprograms, and to match the interface specifications to the task and protected operations that implement them. In this paper we describe the full implementation of task and protected interfaces in the GNAT compiler.

The paper has the following structure: In Section 2 we summarize the main features of Ada 2005 interfaces and focus on those that relate to task and protected types. Section 3 describes the compiler and run-time data-structures related to synchronized types. Section 4 explains how these structures are extended to support dynamic dispatching and how their primitive operations relate to task and protected operations. In addition it also describes the implementation of dispatching calls through synchronized interfaces. Section

SIGAda'05, November 13–17, 2005, Atlanta, Georgia, USA.

5 discusses the implementation of dispatching triggers in selective waits and asynchronous transfer of control, which turned out to be the most complex aspect of the implementation. Section 6 analyzes the cost of this new language constructs. We close with a short discussion on the power of this new language feature and the bibliography.

2. 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 [6]. 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)* —this term is used in the Ada 2005 terminology to refer to the interfaces implemented by a tagged type [1].

```
package Pkg1 is
  type I1 is interface;                    -- 1
  procedure P (A : I1) is abstract;
  procedure Q (X : I1) is null;

  type I2 is interface and I1;             -- 2
  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 Pkg1;
```

The interface *I1* defined at –1– has two subprograms: the abstract subprogram *P* and the null subprogram *Q* (null procedures are described in AI-348 [9]; they behave as if their body consists solely of a *null_statement*). The interface *I2* defined at –2– has the same operations as I1, plus operation *R*. 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 [3, 4] 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
     R (I2'Class (Obj));                   -- 2
  else
     P (Obj);                              -- 3
  end if;
  I1'Write (Stream, Obj)                   -- 4
end Dispatch_Call;
```

The type of the formal *Obj* covers all the types that implement the interface I1, and hence at –3– the subprogram can safely dispatch the call to *P*. However, because I2 is an extension of I1, an object implementing I1 might also implement I2. Therefore at –1– we use the membership test to check at run-time whether the object also implements I2, and then call subprogram *R* instead of *P* (applying a conversion to the descendant interface type I2). 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 provides an additional classification mechanism for interfaces. An interface can be declared to be a limited interface, a synchronized interface, a protected interface, or a task interface [8]. Each one of these imposes constraints on the types that can implement such an interface.

- A task interface can only be implemented 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 tasks or protected types or objects.
- A limited interface can be implemented by tasks types or objects, protected types or objects, or by 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).

In order to combine semantically interfaces with concurrency, we must specify how interface operations are implemented by means of task and protected operations. In particular, we must describe how tasks and protected entries can be said to implement an interface operation. Once this mapping is established, we can treat calls to interface operations as dispatching operations that execute entry calls. (the mapping of protected subprograms and task subprograms onto interface operations is more intuitive and requires less discussion). Using the Ada 2005 object.operation notation [7] we can extend our previous examples as follows:

```
package Pkg2 is
  type I1 is limited interface;            -- 1
  procedure P (Obj  : in out I1) is abstract;
  procedure Q (Obj  : access I1) is null;

  type I2 is synchronized interface and I1; -- 2
  procedure R (Obj  : in out I2;
               Data : in Integer) is abstract;

  type I3 is task interface and I2;        -- 3
  procedure S (Obj  : in out I3) is abstract;
```

```
   type I4 is protected interface;           -- 4
   procedure T (Obj  : in out I4) is abstract;

   task type Tsk is new I3 with              -- 5
      entry P;
      entry R (Data : in Integer);
      ...
   end T1;
   procedure S (Obj  : in out Tsk);          -- 6
   --  Q inherited as a null procedure       -- 7
end Pkg2;
```

At –1– we define a limited interface with an abstract primitive and a null primitive. At –2– we define a synchronized interface that is a derivation if I1. At –3– we define a task interface that is a derivation of the synchronized interface I2. The general rule is that we can compose two or more interfaces provided that we do not mix task and protected interfaces and the resulting interface must be not earlier than any of the ancestor interfaces in the following hierarchy: limited, synchronized, and task/protected. Unlike tagged types, we cannot derive a task or protected type from another task or protected type. So the derivation hierarchy can only be one level deep once we declare actual task or protected types.

At –5– we define a task type that implements the task interface I3 and all its ancestor interfaces (we can also declare a single task or protected object that implements interfaces [10]). Therefore task type *Tsk* must implement all of the abstract primitives of the interfaces concerned. This can be done in two ways, either by declaring an entry or protected operation in the specification of the task or protected type (see the entry declarations at –5–) or by declaring a primitive subprogram in the same declarative part (see –6–). In accordance with the Ada 2005 rules concerning object.operation notation, an entry matches a primitive operation of a progenitor interface if it has the same signature, excluding the first (controlling) argument, which is implicitly the task type itself. Finally, at –7– we note that a null primitive operation can be inherited or overridden as usual. Having declared a number of types implementing a given interface we can dispatch to the various operations in the usual way.

To simplify the presentation, we use the term *synchronized interface* in the rest of this paper. It will be clear that the details that follow apply with little change to the other kinds of interfaces that deal with synchronized types.

3. SYNCHRONIZED TYPES AND CORRESPONDING RECORDS

At first sight the transformation of tasks and protected types into tagged entities would seem to require large changes to the run-time environment, and therefore a major upheaval in the architecture of the system. Fortunately, it turns out that (with one significant exception) all changes are concentrated in the front-end of the compiler, and that once the general dispatching structures for interfaces are implemented, very few changes to run-time structures are required. This proves once again the old adage that every problem in Computer Science can be solved with one additional level of indirection.

The GNAT compiler associates with each synchronized type a *corresponding_record* type. At run-time, every task or protected object is represented by an instance of its corresponding_record. In the case of a protected type, the corresponding_record holds the private data (as defined in the protected definition) and the required lock structure. In the case of a task, the corresponding_record holds principally a pointer to the TCB and other related dynamic information. To implement interfaces and dynamic dispatching, the first step is to make the corresponding_records into tagged types, and to add where necessary wrappers that map the primitive operations of these tagged records into the corresponding task entries and protected operations. The resulting implementation is efficient, and adds at most the cost of one or two indirect calls to a synchronized operation. The semantic complications arise from the matching of the interface operations to task and protected operations, the creation of the proper wrapper functions, and the inevitable visibility problems that arise with the introduction of additional overloaded names. At run-time, an interface operation dispatches through the dispatch table(s) of the corresponding_record. The corresponding primitive operation end up invoking the desired entry or protected operation.

The corresponding_record type (CRT) plays therefore a central role between object-oriented features and synchronization features. The CRT has primitive operations that implement the interface operations. The bodies of these primitive operations are simply wrappers for the actual task or protected operations. The compile-time expansion consists principally in the construction of these primitive operations of the CRT. To understand the details, we must first discuss the way protected and task operations are translated in the absence of interfaces. We refer to this process as *expansion*, because it can be described fairly accurately as a source transformation that is target- independent. In what follows, the result of specific expansion actions is written in quasi-Ada, even though in the compiler it corresponds to a tree transformation..

4. EXPANSION OF SYNCHRONIZED PRIMITIVE OPERATIONS

The following sections describe the expansion of subprograms and entries that are primitive operations of synchronized interfaces. We also describe the basic run-time support required to give support to interfaces (for further information see [12]).

4.1 Expansion of protected subprograms

GNAT treats protected subprograms rather differently from non-protected ones. For each protected routine, the compiler generates two operations: a protected and an unprotected version, that are invoked depending on the current synchronized context [13].

The unprotected version is used in internal calls from one protected subprogram to another within the same protected object. Since in this case we are in the same synchronized environment, there is no need to recapture the already seized locks. The body of the unprotected version simply executes the statements of the body of the original protected subprogram.

The protected version on the other hand is used for external calls on the object. At the point of invocation the subprogram first seizes the locks of the object, then performs a call to the unprotected version, and finally releases the locks on exit. As a result, an external call typically results in several system calls for lock management.

If the protected type *Prot* implements a synchronized interface *SI*, and the protected operation implements one of the operations of SI, the compiler now generates a third body, which is a primitive operation of the corresponding_record of Prot. This primitive operation has the same signature as the interface operation; its body simply invokes the protected version, given that interface dispatching calls are always external calls (by definition it is not in general possible to determine the target object of the call). To illustrate, consider the following protected type:

```
type Synch_Interface is synchronized interface;
procedure Bar (Obj : in out Synch_Interface);

protected type Prot is new Synch_Interface with
   procedure Bar;
end Prot;

protected body Prot is
   procedure Bar is
   begin
      Put_Line ("Hello from Prot");
   end Bar;
end Prot;
```

Here are the different versions of *Bar* generated by the compiler during code expansion. The compiler generates distinct internal names for these, to simplify debugging. The encoding used below is self-explanatory.

1. Unprotected version.

```
procedure BarU
  (_object : in out Prot_CRT) is
begin
   Put_Line ("Hello from Prot");
end BarU;
```

2. Protected version.

```
procedure BarP
  (_object : in out Prot_CRT) is
begin
  <capture _object's locks>
  BarU (_object);
  <release _object's locks>
end BarP;
```

3. Primitive operation of the CRT.

```
procedure Bar
  (_object : in out Prot_CRT) is
begin
  BarP (_object);
end Bar;
```

Once the primitive operations (also called primitive wrappers) are created, the compiler handles them like other primitive operations of a tagged type, and creates slots for them in the dispatch table of the CRT, at the appropriate position.

4.2 Expansion of protected and task entries

Entries undergo a more complex transformation than protected subprograms, but the generation of the corresponding primitive wrappers for their CRT is analogous. Consider the following declaration and body:

```
task type Tsk is new Synch_Interface with
   entry Bar;
end Tsk;

task body Tsk is
begin
   accept Bar do
      Put_Line ("Hello from Tsk");
   end Bar;
end Tsk;
```

For entry Bar, GNAT generates the following wrapper:

```
procedure Bar
  (_object : in out Tsk_CRT) is
begin
  Tsk (_object).Bar;  -- Object.Operation notation
end Bar;
```

Note that we perform a type conversion from the corresponding_record type to the original synchronized type. As a consequence, the expansion of this call within the wrapper results in the correct expansion of an entry call: parameter transfer between stacks, context switching, etc.

4.3 Dispatching through synchronized interfaces

The data structures involved in dynamic dispatching are identical for all types that implement interfaces. A primary dispatch table holds pointers to all primitive operations of the type. Secondary dispatch tables are created for each one of the interfaces that the type implements. Each secondary dispatch table holds pointers to the operations of the type that implement the corresponding operations of the interface. Full details can be found in [12].

```
type Regular_Interface is interface;
type Synch_Interface is synchronized interface;
```

These two types appear in the expanded tree as:

```
type Regular_Interface
  is abstract tagged null record;
type Synch_Interface
  is abstract tagged limited null record;
```

Note that the underlying representation of Synch_Interface is labeled as "limited", which indicates that it can be implemented by a limited type.

```
protected type Prot and Synch_Interface is ...
task type Tsk and Synch_Interface is ...
```

The corresponding_record types are declared as:

```
type Prot_CRT is tagged limited record ...
type Tsk_CRT  is tagged limited record ...
```

The protected subprograms and entries of protected and task types do not appear directly in the runtime structures of these types. At the point when the types are elaborated and the dispatch tables generated, GNAT collects the primitive wrappers and the dispatching versions of protected subprograms that override some interface-level operation. The entries in the dispatch table are then set to contain their addresses. Referring back to our previous example, Figure 1 contains the dispatch table of type Prot.

In the case of type Tsk, the dispatch table entry for Bar will point to the primitive wrapper. In this fashion the existing interface machinery present in GNAT for regular tagged types is reused without further modification.

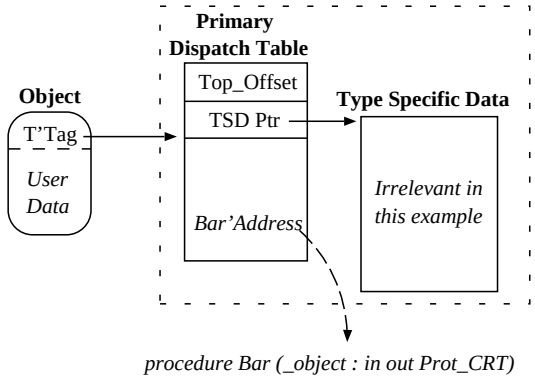

Figure 1: Run-Time Data Structure

```
declare
  procedure Dispatch_On_Bar
    (Any_Synch : in out Synch_Interface'Class) is
  begin
    Any_Synch.Bar;  -- Dispatching call
  end Any_Bar;

  Obj_1 : Prot;
  Obj_2 : Tsk;
begin
  Obj_1.Dispatch_On_Bar;
  Obj_2.Dispatch_On_Bar;
end;
```

Both of these dispatching calls will examine the dispatch table slot for Bar and will invoke the subprogram designated by the contents of the slot. In the case of Prot, it is the dispatching version of the protected procedure. The call will then invoke the protected version, which in terms will seize the lock of Obj_1, call the unprotected version and output "Hello from Prot", and finally release the lock. The dispatching call on Obj_2 will in turn find the generated primitive wrapper, invoke it and output "Hello from Tsk".

5. DISPATCHING IN SELECT STATEMENTS

As a final extension of the interface machinery, AI-345 [5] allows dispatching interface operations to appear in *entry_call_alternatives* of asynchronous, conditional and timed selects. This capability complicates the expansion of the already intricate handling of these constructs. The complications come from the fact that in normal (non-dispatching) cases, the expansion of the enclosing construct (selective wait or ATC) depends on whether the target of the triggering call is a task or protected object. If the call is to an interface operation, the compiler cannot determine which variety of expansion to use, and must make provisions to:

1. Determine at run-time whether the target object is a protected type, a task type, or a tagged type.

2. Choose the remaining actions of the construct according to the result of this determination. This means that the expanded code must contain provisions for all three possibilities.

Consider the following fragment:

```
select
  Obj_1.Dispatch_On_Bar;
  Put_Line ("Call dispatched");
or
  delay 1.0;
  Put_Line ("Timer expired");
end select;
```

The semantics are intuitive: if the call dispatches to a conventional (non-protected) subprogram, it is *accepted* at once, and the delay is ignored. Otherwise, the usual timed entry call semantics apply.

In order to solve problem 1) we must extend the dispatch table structures to include two new tables, located within the *Type Specific Data* of synchronized types that implement a limited interface. One table contains the *operation kind* (which indicates whether a given entry corresponds to asubprogram, protected subprogram, protected entry, or task entry). The other table, which will not concern us further here, captures the index of an entry (this is used to determine the position of the queue for this entry in the CRT). Figure 2 contains an extended view of the dispatch table for Prot and Figure 3 contains the dispatch table for Tsk.

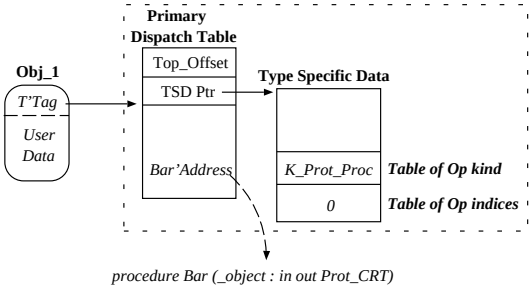

Figure 2: Run-Time Data Structure: Prot

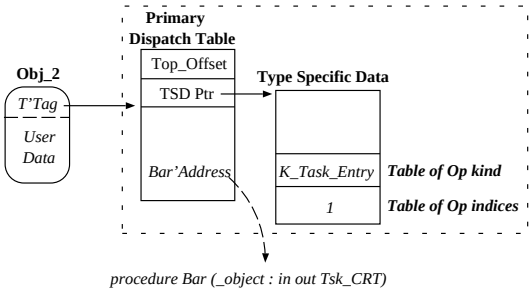

Figure 3: Run-Time Data Structure: Tsk

Four dispatching predefined primitive operations are used to both determine the operation kind of a call and invoke a library runtime operation if the call dispatches to an entry. These four predefined operations are generated for every limited interface as abstract procedures and are implemented by any type that implements the interface. In other words, they are implicit intrinsic operations.

```
_disp_asynchronous_select
 (Obj : <type>;           -- object
  S   : Integer;          -- prim. operation DT slot
  P   : System.Address;   -- wrapped parameters
  B   : out Communication_Block;
  F   : out Boolean);     -- call status flag
```

```
_disp_conditional_select
 (Obj : <type>;           -- object
  S   : Integer;          -- prim. operation DT slot
  P   : System.Address;   -- wrapped parameters
  C   : out Prim_Op_Kind; -- callable entity kind
  F   : out Boolean);     -- call status flag

_disp_get_prim_op_kind
 (Obj : <type>;           -- object
  S   : Integer;          -- prim. operation DT slot
  C   : out Prim_Op_Kind);-- callable entity kind

_disp_timed_select
 (Obj : <type>;           -- object
  S   : Integer;          -- prim. operation DT slot
  P   : System.Address;   -- wrapped parameters
  D   : Duration;         -- delay time
  X   : Integer;          -- delay mode
  C   : out Prim_Op_Kind; -- callable entity kind
  F   : out Boolean);     -- call status flag
```

There is one operation per select kind, with asynchronous select utilizing a helper routine. The body for *_disp_timed_select* created for Prot reads as follows:

```
procedure _disp_timed_select
  (Obj : <type>;           -- object
   S   : Integer;          -- prim. operation DT slot
   P   : System.Address;   -- wrapped parameters
   D   : Duration;         -- delay time
   X   : Integer;          -- delay mode
   C   : out Prim_Op_Kind;-- callable entity kind
   F   : out Boolean)      -- call status flag
is
   I : Integer;

begin
   C := Get_Prim_Op_Kind (Obj._tag, S);      -- 1

   if C = K_Procedure                        -- 2
     or else C = K_Protected_Procedure
     or else C = K_Task_Procedure
   then
      F := True;
      return;
   end if;

   I := Get_Entry_Index (Obj._tag, S);       -- 3

   Timed_Protected_Entry_Call                -- 4
    (Obj._object'unchecked_access, I, P, D, X, F);
end _disp_timed_select;
```

1. The first step of the execution is to access the table of primitive operation kinds to determine whether the call dispatches to a procedure or to an entry.

2. If the call dispatches to any kind of procedure, treat it as if an entry call has accepted and executed.

3. Otherwise it is known that the call dispatched to an entry and it is safe to retrieve its index from the index table.

4. Finally, perform a call to the appropriate runtime routine to execute a timed entry call.

Note that *Get_Prim_Op_Kind* and *Get_Entry_Index* are not dispatching calls, those are simply invocations of library routines whose purpose is to examine the Type Specific Data of Obj. A possible optimization is to merge the two calls or to completely remove the invocation of Get_Prim_Op_Kind and use a preset index value to denote a procedure.

When a select statement is expanded, GNAT generates a dispatching call to one of these subprograms instead of directly performing the original dispatching call. Referring back to our timed select example, its expansion is as follows:

```
declare
  B  : Boolean := False;                      -- 1
  C  : Prim_Op_Kind;                          -- 2
  DX : Duration   := ...;  -- delay time
  M  : Integer    := ...;  -- delay mode
  P  : Parameters := (Param1 .. ParamN);      -- 3
  S  : constant Integer := <DT_Pos>;          -- 4

begin
  _disp_timed_select                          -- 5
    (Obj_1, S, P'address, DX, M, C, B);

  if C = K_Protected_Entry                    -- 6
    or else C = K_Task_Entry
  then
    Param1 := P.Param1;
    ...
    ParamN := P.ParamN;
  end if;

  if B then                                   -- 7
    if C = K_Procedure
      or else C = K_Protected_Procedure
      or else C = K_Task_Procedure
    then
      Obj_1.Dispatch_On_Bar;                  -- 8
    end if;

    Put_Line ("Call dispatched");             -- 9
  else
    Put_Line ("Timer expired");              -- 10
  end if;
end;
```

The parameters and body of the operation have the following function:

1. The boolean flag B is used to determine whether the runtime entry call has been accepted.

2. The primitive operation kind to be retrieved from the table in the type specific data of the actual type.

3. The parameters in the call are collected into a record that is passed to the stack of the callee (in the case of a task).

4. The position of the operation in the dispatch table is retrieved.

5. Call to the appropriate dispatching select handler. This call will retrieve the operation kind at location *S*. If it is an entry, it will also retrieve its entry index and execute it via a runtime library call.

6. If the dispatching select handler determines that the call is to an entry, the out-mode parameters must be copied back from the communication record and assigned to the original actuals.

7. This branch is executed if the entry call has been accepted or the dispatching select handler found a procedure.

8. Since the operation kind is now known, the original dispatching call may be executed since it is certain that it is a procedure and no runtime support is needed.

9. The triggering statements are executed only when the entry call is accepted or the dispatching select handler found a procedure.

10. The delay statements are executed only if the entry call was not accepted.

Conditional selects are handled in a similar fashion to timed selects, the only difference behind is that the predefined primitive operation invokes a different runtime library routine. Asynchronous selects are the most complicated of the three and require an additional helper routine. The full description of the mechanism can be found in *exp_ch9.adb* (sources available in the FSF repository [14]). The following pseudo-code summarizes all the actions performed in a dispatching ATC:

```
<invoke _disp_get_prim_op_kind and store the
  operation kind in C>

if C is a protected entry then
  <define a finalization _clean routine to cancel
   the protected entry call>
  <defer_abort>
  <invoke _disp_asynchronous_select>
  <copy the parameters of the call from the
   wrapping record type to the actual parameters>

  if the entry call is enqueued
     <execute the abort statements>
  <perform finalization via _clean>

  if an abort signal exception occurs
     <undefer abort>
  if the entry call is not canceled
     <execute the triggering statements>

elsif C is a task entry then
  <define a finalization _clean routine to cancel
   the task entry call>
  <defer abort>
  <invoke _disp_asynchronous_select>
  <copy the parameters of the call from the
   wrapping record type to the actual parameters>
  <undefer abort>
  <execute the abort statements>
  <perform finalization via _clean>

  if an abort signal exception occurs
     <undefer abort>
  if the entry call is not canceled
     <execute the triggering statements>

else
  <execute the original dispatching call since
   it will dispatch to a procedure>
  <execute the triggering statements>
end if
```

5.1 Other interface operations

The attribute 'Callable and 'Terminated, as well as the operation Abort, are applicable to a task interface. Their implementation is trivial, and requires no dispatching machinery: any type that implements such an interface must be a task, and the attribute or operation will be directly applicable to the access type that denotes the task at runtime.

6. PERFORMANCE

Dispatching calls through abstract interface types are performed in GNAT in constant time [12]. Dispatching select constructs are dependent on the performance of the runtime library. Regardless of that fact, the overhead involved in running those routines is at least one and at most two dispatching calls. Asynchronous dispatching selects have a constant cost of two dispatches: one to invoke *_disp_get_prim_op_kind* and one to execute the original dispatch call or *_disp_asynchronous_select*. The lower bound on performance of conditional and timed dispatching selects is one dispatching call. This case occurs whenever the call to *_disp_conditional_select* or *_disp_timed_select* dispatches to an entry. In this situation the runtime library routine is invoked from within those two routines. An upper bound of two dispatching calls is observed whenever the call dispatches to a protected subprogram. In this situation *_disp_conditional_select* or *_disp_timed_select* simply return after determining the primitive operation kind of the call and execute the original dispatching call.

7. CONCLUSION

Interfaces are one of the most significant features of Ada 2005. The combination of inheritance, interfaces and synchronization (an elusive goal of programming languages for close to two decades) is clean and intuitive. It is gratifying to find out that the implementation of this extended facility is not overly complex, and that it can be integrated in the architecture of an existing compiler.

Over the last year the GNAT development team has been working on the implementation of the most important Ada 2005 issues [11]. This paper completes the description presented in [12], on the GNAT implementation of abstract interface types. 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 cannot 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 the dedicated and enthusiastic members of AdaCore, and the myriad supportive users of GNAT whose ideas, reports, and suggestions keep improving the system.

8. REFERENCES

[1] Ada Rapporteour Group. *Annotated Ada Reference Manual with Technical Corrigendum 1 and Amendment 1 (Draft 13): Language Standard and Libraries*. (Working Document on Ada 2005).

[2] 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.

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

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

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

[6] Ada Rapporteour 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.

[7] Ada Rapporteour Group. *Object.Operation Notation*. Ada Issue 252, Available at http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AIs/AI-00252.TXT.

[8] Ada Rapporteour Group. *Protected and Task Interfaces*. Ada Issue 345, Available at http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AIs/AI-00345.TXT.

[9] Ada Rapporteour Group. *Null Procedures*. Ada Issue 348, Available at http://www.ada-auth.org/cgi-bin/cvsweb.cgi/AIs/AI-00348.TXT.

[10] Ada Rapporteour 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.

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

[12] 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.

[13] J. Miranda. *A Detailed Description of the GNU Ada Run-Time*. Free book available at http://www.iuma.ulpgc.es/users/jmiranda/gnat-rts/index.htm, 2002.

[14] Free Software Foundation. *GNAT Sources Repository*. http://www.gnu.org/software/gnat/gnat.html

[15] AdaCore. *GNAT Academic Program*. http://www.adacore.com/academic_overview.php