Features of the Gnu Ada Runtime Library

Draft

E.W. Giering   Frank Mueller   T.P. Baker[1]
Department of Computer Science
Florida State University
Tallahassee, FL 32306-4019

Internet: giering@cs.fsu.edu   Phone: (904)644-3441

As submitted to TRIAda '94

[1] Partially funded by the Ada Joint Program Office under the Ada Technology Insertion Program, through the U.S. Army CECOM.

Features of the Gnu Ada Runtime Library

Draft

ABSTRACT

The GNu Ada Runtime Library (GNARL) is being developed to support Ada 9X tasking for the Gnu NYU Ada Translator (GNAT). Together, they form a portable, freely distributable Ada 9X translation system. GNARL and GNAT communicate through a well-defined procedural interface, facilitating their independent development.

Among the design goals of this translation system are portability, interoperability with other languages (in particular C), efficiency, and user extensibility. This paper provides an overview of how the GNARL supports these goals.

1 INTRODUCTION

Ada requires a more complex runtime environment than most other languages; in particular, it requires support for multiple tasks. The GNu Ada Runtime Library (GNARL) is being developed at Florida State University to provide tasking support for the Gnu NYU Ada Translator (GNAT), an Ada 9X compiler and associated runtime system (RTS) being developed separately at New York University [10]. This separate development is facilitated by a well-defined procedural interface between code generated by the compiler for tasking constructs and the GNARL, described in the GNARL interface document [5].

The Gnu NYU Ada Translator (GNAT) adds to the suite of languages supported by the the Gnu C Compiler (`gcc`), which already includes compilers for C, C++, and Objective C. Like the other tools, GNAT and its associated runtime, including GNARL, are provided in source without charge, and can be freely copied under the terms of the Gnu software license agreement.

GNARL is intended to support GNAT in achieving a number of design goals. Chief among these are:

- Portability
- Efficiency
- Language interoperability, in particular between Ada and C

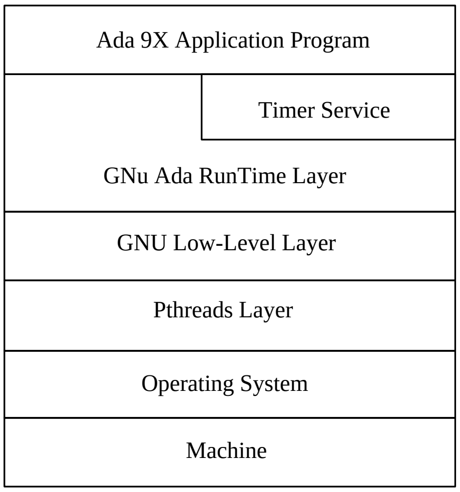

Figure 1: GNARL componants

- User extensibility

The remainder of this paper describes the features of GNARL that aid GNAT in achieving these goals. Two of these features are described in greater detail: runtime elaboration from other languages as supported by GNAT, a language interoperability issue; and user extensibility through user-defined time types.

2 PORTABILITY

To facilitate the porting of tasking support to a new platform, the GNARL is implemented in multiple layers, as illustrated in Figure 1.

Each layer provides all of the tasking-related services required by the next higher layer through a procedural interface. The exception is Timer Service; this forms a layer over the GNARL interface, but applications have access to both the Timer Service and the GNARL. This is explained in more detail in Section 5.

As GNARL and the lower-level library are written in Ada, they can be easily ported to any system that supports the POSIX Threads Extension (Pthreads) once the rest of GNAT has been ported. The GNAT compiler consists of an Ada front end driving the `gcc` code generator. This code generator is designed to be easily retargetable and has been ported to numerous platforms; GNAT and therefore GNARL inherit this portability.

Pthreads is an extension to the POSIX standard for an application program interface to operating systems services. Pthreads provides basic multithreaded services within a single address space. The GNARL uses these services to build tasking services with correct Ada semantics. Pthreads has not yet been approved as a standard, and many platforms will lack a Pthreads implementation. In such cases, it would be possible to port tasking services by reimplementing the GNARL. However, implementing GNARL semantics is relatively complex, and will probably be of interest only to users requiring unusual tasking semantics, or to take advantage of unusual hardware architecture such as multiprocessing or distributed environments. For this reason, an intermediate layer, the GNU Lower-Level (GNULL), has been added.

The GNULL interface is designed to be trivially implementable on a system supporting Pthreads, and easily implementable over one that does not (e.g. a bare machine). The GNULL interface document describes semantics for multiple threads of control that allow the next higher layer (the GNARL) to implement Ada tasking [3, 4]. The current GNULL consists for the most part of selected Pthreads operations, some with simplified semantics. Tasking support can be ported by implementing the much simpler GNULL layer and compiling the unmodified GNARL on the new target.

3 OPTIMIZATION

The GNARL is designed to facilitate the in-line optimization of Ada tasking constructs. The use of task constructs results in the implicit **with** of one or more of the packages that make up the GNARL by the GNAT compiler. Other than this implicit import, GNARL packages are indistinguishable from other application packages. There are no special restrictions on GNARL code. In particular, GNARL subprograms can be named in `Inline` pragmas, resulting in the replacement of implicit calls to these subprograms with the subprogram body.

This should result in somewhat faster code due to the elimination of the subprogram call. However, once the code has been inserted inline, it can be further optimized by the compiler using information about the local environment such as current register contents. This process is further augmented by the inline nature of the GNARL interface. Tasking is implemented with calls to the GNARL interleaved with user code. The only exception to this is task startup, where GNARL executes the task body code from a new thread of control via callback. This inline nature of the GNARL interface is intended to allow local optimizations across the boundaries between the application and the GNARL, in particular when the GNARL calls are expanded inline. This kind of optimization is much less applicable with an interface involving callbacks to user code within the RTS. Each callback point can call one of an arbitrary number of user code sequences, so they cannot be inlined, and it is less likely that local optimizations such as register allocation will be applicable to all of them.

4 LANGUAGE INTEROPERABILITY

This section discusses the interoperability of Ada and other programming languages, such as C or C++, within the context of the GNARL and the code generated by the GNAT compiler. The two main issues are: First, the GNARL is intended to be built over the C-language Pthreads interface. This allows the coexistence of Ada tasks and C threads within a mixed-language environment. Second, the GNARL can be provided as an object library file which can be linked into a mixed-language application, even if the main subprogram is not written in Ada. The correct elaboration of the GNARL can still be guaranteed. The discussion below explains how elaboration can be generalized for arbitrary Ada object libraries with few restrictions.

One of the major features of the GNAT is its interoperability with the other languages, in particular C. Much of this interoperability is achieved through the use of the common code generator. The code generator contains all knowledge of such issues as data representation, register usage, and subprogram calling conventions, so languages can share many basic data types and call one another's subprograms.

There are limits to the interoperability of software packages in general independent of language issues. In particular, signal handling under UNIX limits the interoperability between object libraries. If two libraries which install a handler for the same signal are linked together, only one handler will generally be active. The GNARL installs signal handlers for a number of signals. When a program is linked with GNARL, none of its components should install a handler for these signals.

4.1 COEXISTENCE OF ADA TASKS AND C THREADS

The GNARL provides interoperability with the C programming language through the use of Pthreads. Unlike Ada, the C language does not define multiple threads of control; multithreaded services must be added. Pthreads is a draft standard C language interface to services for creating and manipulating multiple threads of control, and for executing C functions within them. This is intended to allow multithreaded applications to be written portably in C, and as such performs much the same function for C as tasks do for Ada.

This poses a potential problem for an application consisting of C and Ada code. By using the `gcc` back end, the languages are able to call one another. However, if the Ada program creates tasks and the C program creates threads, they must be able to coexist and interact safely. If tasks and threads are implemented separately, each with its own scheduler, there will be no well-defined way for threads and tasks to contend for the processor.

The GNARL is intended to be easily implemented over a C language binding to Pthreads, so that the Pthreads implementation treats tasks as threads. Since Ada tasks are threads, they are scheduled together with the threads created by C code in a well-defined way [7, 6].

4.2 ELABORATION IN A MIXED-LANGUAGE ENVIRONMENT

In a mixed-language environment, an Ada program can call a non-Ada subprogram and vice versa. The Ada portion in this environment requires elaboration of the Ada RTS prior to executing any Ada code to exhibit correct semantic behavior. The following discussion includes several scenarios of language interoperability in conjunction with elaboration of Ada code. In addition, an outline is given for creating Ada object libraries which can be interfaced by non-Ada languages.

4.2.1 CALLING ANOTHER LANGUAGE FROM ADA

When an Ada program calls non-Ada subprograms, the issue of elaboration is much the same as in a pure Ada environment. Library units which require elaboration are elaborated prior to the execution of the main subprogram written in Ada (ARM 10.2(10) [1]). For the GNAT compiler, the elaboration of the GNARL includes dynamic storage allocation and initialization of global data. It can generally be assumed that the stack of the environment task is already available at elaboration time and that storage allocation routines are initialized (if not even explicitly initialized by the Ada RTS).

The GNAT compiler handles elaboration in general through the `gnatbind` tool which generates code for the Ada environment task. This tool is called after all units required for the program have been compiled. The tool then determines an elaboration order for the required Ada packages and generates a call for the elaboration of each package specification and each package body in the determined order within the environment task. The elaboration calls are followed by a call to the main subprogram which is followed by calls to finalization routines, if required.

4.2.2 CALLING ADA FROM ANOTHER LANGUAGE

The `pragma Export` in the Ada 9X Annex (ARM M.1 [1]) allows Ada entities (in particular functions and procedures) to be exported under a certain linker name. When a non-Ada program calls an exported Ada subroutine, the Ada RTS has to be elaborated prior to the execution of the called subroutine.

Elaboration of Ada packages can be achieved by coupling the code generation of the compiler with the linker. Consider other languages which require elaboration (and finalization), such as C++, which has constructors and destructors [8]. A static storage type constructor is a routine which is called prior to the main subprogram. Let us assume a format for object files which allows the specification of elaboration and finalization code in special code sections.[2] In such an environment, the Ada compiler emits elaboration code in the elaboration section just as the C++ compiler emits calls to static constructor functions in the same section. It then becomes the linker's responsibility to statically link the object files and produce an executable where first the elaboration section is executed, followed by a call to the main subprogram, and then to the finalization code.

Involving the linker in the elaboration procedure affects the compliance with Ada semantics as follows:

- It does not incur any runtime overhead for elaboration checks of preelaborated units.

- A legal elaboration order with respect to Ada can be determined by checking the dependences between elaboration sections of linkable units at link-time. This assumes that the elaboration for a package specification resides in a different file than the elaboration of the package body, thereby allowing cross-dependences between the elaboration of specs and bodies.[3] The elaboration sections of each Ada package are then placed in the determined order.

- If an exception-catching mechanism is installed during elaboration, the semantics of exception propagation is preserved as follows. If an Ada subroutine raises an exception, the exception may propagate into the non-Ada code. If non-Ada code cannot handle the exception, it will propagate to the main subprogram. The main subprogram, al-

[2] Under Solaris 2.x, the `.init` and `.fini` sections are code sections which are invoked prior to and after the call to the main subprogram respectively [9]. The corresponding C pragmas are `#pragma init` and `#pragma fini` respectively.

[3] A cross-dependence exists if, for two packages A and B, the elaboration of body A depends on the elaboration of spec B and the elaboration of body B depends on the elaboration of spec A. A legal elaboration order would be spec A, spec B, body A, body B. The GNAT compiler currently emits only one object file for both spec and body of a package. The elaboration code is generated by `gnatbind` tool. To take advantage of the elaboration and finalization sections, the GNAT compiler would have to emit separate object files for spec and body of a package to allow proper ordering at link time in case of cross-dependences.

though non-Ada code, is embedded in the Ada environment task as part of the elaboration which then reports the exception and terminates the program. An alternate method is discussed later.

In some cases, however, the Ada RTS may not be required at all when an Ada subroutine is called. If all runtime checks are suppressed, no tasking features are used, and exceptions can be ruled out at the design level of some Ada unit, the linker can produce an executable without the Ada RTS, thus reducing both code size and elaboration overhead.

4.2.3 STATIC OBJECT LIBRARIES

Handling elaboration by cooperation between the compiler and linker provides an opportunity to create object libraries which contain a collection of exported interfaces for use by other languages. Object libraries are supported by many operating systems, for example UNIX. During compilation of Ada packages, the GNAT compiler generates an object file for each package. An object library can simply be created by archiving a set of object files in a library. The object library can be used by linking it with a main program written in any language. Generally, the GNARL will have to be linked with the program as well.

When Ada object libraries are linked with non-Ada programs, the elaboration is handled by executing the elaboration sections which are generated by the compiler. If multiple Ada libraries are linked, the elaboration order may depend on the properties of the linker. Under Solaris 2.x, for example, libraries are elaborated in the opposite order in which they are specified for linking [9]. In this case, the GNARL should always be specified last for linking since other Ada libraries may use runtime features during elaboration. The elaboration dependences of an Ada object library on other Ada libraries should always be sufficiently documented.[4]

4.2.4 DYNAMIC OBJECT LIBRARIES

Using the linker approach and object libraries as discussed previously, the concept of static object libraries can be extended to dynamic libraries. Dynamic libraries allow the code portion of a library to be shared by different processes, yet a separate data portion is created for each process [9]. This can be useful for reducing the memory requirements of programs in a multi-tasking environment where frequently executed code is shared, for example the GNARL in an Ada environment. It also provides a simple means to revise libraries since the latest version of a named library is chosen at load time or even during program execution. In addition, a dynamic library may depend on other dynamic libraries which will cause them to be loaded (and elaborated) transitively in the order of dependence. For example, a dynamic Ada library may depend on the Ada runtime library. In this case, the latter library will be elaborated prior to the former library.

Dynamic libraries can be linked statically or dynamically:

Static Linking: The names of dynamic libraries are provided at link time and the library is selected and loaded at load time. The elaboration (finalization) of a statically linked dynamic library is performed prior (after) executing the main subprogram.

Dynamic Linking: This option provides an opportunity to delay the loading of dynamic libraries into memory until they are needed ("lazy linking"). The library is loaded during program execution. The elaboration (finalization) of a dynamicly linked dynamic library is performed prior to (after) the first (last) reference to a unit in the library. A reference to a dynamic library consists of a system call to open or close the library.

An application can reduce the code size and startup time for the average case if certain infrequently executed code is loaded as a dynamic library upon conditional execution.

When an exception is raised in some Ada code which was dynamically linked and no exception handler is found in the invocation sequence of Ada routines, the exception might propagate outside the Ada frames into non-Ada routines. Some languages like C++ also have exception handling facilities and may catch the Ada exception if the exception handling facility is compatible across languages. But if the exception propagates into non-Ada frames and is not handled, the exception will propagate past the main subprogram. Since no Ada elaboration may have taken place prior to executing the main subprogram, no exception handler will be found. The exception propagation eventually tries to access an invalid stack frame causing a signal handler to execute which then shuts down the entire program.

One exception to the correct handling of dynamic linking is the handling of preelaborated packages. If an object library contained preelaborated packages and was dynamically linked, it would violate the Ada restriction that preelaborated packages may not impose any dynamic elaboration checks. Dynamic linking of any library will force any externals declared within this library to be resolved at the library load point. This can clearly be viewed as an elaboration check which in-

[4] It would be possible to determine the elaboration order of libraries at link time and to reject circular elaboration dependences. Such a tool will be subject of future work.

curs runtime overhead. Preelaborated packages must therefore not be linked dynamically.

Another problem is caused by side-effects as part of elaboration code. Consider some package elaborating which writes to a file. With static linking, this elaboration code was executed prior to the main subprogram and thus exhibits correct Ada semantics. But with dynamic linking, this write operation occurs during execution of the main subprogram at the library load point. This clearly violates Ada semantics. Dynamically linked libraries must there not contain elaboration code with side-effects. The compiler support could be extended to include an implementation-defined `pragma Elaborate_Without_Side_Effect` on a package. If this pragma was implemented, the compiler could check for side-effects upon encountering the pragma and would report an error if side-effects exist.

Overall, the GNAT environment provides sufficient means for language interoperability and provides the facilities for creating object libraries which greatly enhances the ability for non-Ada languages to interface with Ada code. The GNARL can be provided as a (preferably dynamic) object library and can be linked with non-Ada applications. For dynamically linked libraries, an implementation-defined pragma can be provided to ensure that elaboration does not have any side-effects. The remaining support for language-interoperability does not require implementation-defined extensions to the Ada language.

5 USER EXTENSIBILITY

GNARL, like GNAT, is provided as source code, which provides users with the opportunity to modify and extend the system to meet special needs. In addition, the layered structure of the GNARL isolates different aspects of tasking (time, Ada tasking, threads), making modifications to these different aspects simpler and safer.

The timer service layer was added to allow arbitrary numbers of implementation defined time types without modification to the GNARL interface. This has had the side effect of allowing definition of time types by the user that can then be used in Ada tasking constructs. This section discusses this aspect of GNARL extensibility in more detail.

5.1 ADA 9X TIME TYPES

In Ada 83, delays are implemented using **delay** clauses, which provide timeouts for a number of dissimilar Ada task constructs implemented using dissimilar tasking interfaces. The delay interval is always specified with an expression of the predefined type `Duration`. This simplifies the interface to tasking operations; any operation which can time out (entry calls, select statements, and delay statements) was provided with a `Duration` parameter to specify the timeout interval (if any). For example, in the Ada 83 Common Ada Run-Time System (CARTS) [2], the timed entry call

```
select
   T.E;
or
   delay 5.0;
end select;
```

would be translated into a call to `Call_Timed`:

```
Call_Timed(T, E, null, 5.0, Rendezvous_Successful);
```

Here the **delay** clause has contributed the timeout parameter `5.0` to this call.

In addition to the **delay** clause, Ada 9X introduces the **delay until** clause, which can appear wherever the **delay** clause can. This allows the specification of an absolute timeout, specified using a *time type*. Ada 9X defines one time type, `Ada.Calendar.Time`, in the core document, and another, `System.Real_Time.Time`, in the Annex H (Real-Time Systems). In addition, the current draft Ada 9X Reference Manual (ARM) states that the argument to a **delay until** statement may be any implementation-defined time type (ARM 4.0 9.6(4)) [1].

This complicates the original Ada 83 interface scheme considerably. Some means must be provided for passing an argument of either duration or any of an arbitrary number of time types to each operation that can have a timeout.

5.2 INDIVIDUAL DECLARATION OF EACH TIME TYPE

One possibility is to allow each RTS operation requiring a timeout to take any of the implementation-defined time types. This could be done by providing each such operation with parameters of duration and of each time type. For example, the operation for a timed entry call might be:

```
procedure Call_Timed
  (...;
   TO_Duration : Duration;
   TO_Calendar_Time : Ada.Calendar.Time;
   TO_Real_Time : System.Real_Time.Time;
   TO_Impl_Def_1 : Imple_Def_1.Time;
   TO_Impl_Def_2 : Imple_Def_2.Time;
   ...);
```

Alternatively, a set of overloaded procedures could be provided for each operation, one for **Duration** and one for each time type:

```
procedure Call_Timed(...T : Duration...);
```

```
procedure Call_Timed(...T : Ada.Calendar.Time...);
procedure Call_Timed(...T : System.Real_Time.Time...);
procedure Call_Timed(...T : Impl_Def_1.Time...);
procedure Call_Timed(...T : Impl_Def_2.Time...);
```

Both of these methods require a different runtime interface definition for each set of time types, so that one interface no longer can cover all implementations. The result is an interface scheme instead of an interface. The compiler and the RTS need to be modified for each new time type, and for each change to a time type representation.

Time can also be defined as a single variant record type, with a variant for each time type. This avoids an explicit mention of each time type in the definition of each RTS operation that needs a timeout, at the cost of the runtime overhead of determining the time type from the record discriminant. However, this still requires the RTS interface describing the variant type, as well as the internal RTS code, to change whenever a time type is added or modified. An untyped interface would also be possible, with the RTS coercing the single time type into the specified time type. This solution would eliminate the need to modify the interface, but not the need to modify the internal code. It is also dangerous, particularly in a system in which the RTS is as open to modification as the GNARL.

5.3 DECLARATION OF A SINGLE TIME TYPE

Another possibility is to define some form of polymorphic type which can be used to represent a value of any time type. One possibility is to define a single abstract time type:

```
package Abstract_Time is
   type Time is abstract tagged null record;
   function Clock return Time is abstract;
   function "<"(Left, Right: Time)
      return Boolean is abstract;
   type Asynchronous_Delay is private;
   procedure Asynchronous_Delay_Until
     (T: Time;
      AD:out Asynchronous_Delay) is abstract;
   procedure Cancel_Delay
     (AD: in out Asynchronous_Delay) is abstract;
end Abstract˙Time;
```

All time types would be defined by extending this abstract type:

```
with Abstract_Time;
package Ada.Calendar is
   ...
   type Time is private;
   ...
   function Clock return Time;
   ...
   function "<"(Left, Right: Time) return Boolean;
   ...
private
  type Time is new Abstract_Time.Time with record
     implementation-defined
  end record;
end Ada.Calendar;

with Abstract_Time;
package Ada.Calendar.Delays is
   ...
   procedure Asynchronous_Delay_Until
     (T: Time;
      AD:out Asynchronous_Delay);
   ...
end Ada.Calendar.Delays;
```

The intent is that all GNARL operations involving timeout would take an argument of a class-wide type of the abstract `Time` type:

```
procedure Call_Timed
  (...
   T : Abstract_Time.Time'Class;
   ...);
```

This allows `Call_Timed` to take an argument of any type derived by extending `Abstract_Time`. These operations would then be implemented in terms of the abstract operations on this type, which would dispatch to the correct overriding operations for the extended type.

Delays pose a problem, particularly asynchronous delays, where the calling task has to be interrupted when the timeout occurs. In the proposed `Abstract_Time` package, all delays are accomplished by the **Asynchronous_Delay_Until** operation. This operation interrupts the current task when the specified delay has completed by raising a special exception. This exception can then be caught at the point where execution is to resume. A simple delay statement could be implemented using this operation in conjunction with a suspension (such as a deadlocked entry call):

```
task body T1 is
...
   declare
      T : Ada.Calendar.Time := ...;
      AD : Asynchronous_Delay;
   begin
      Asynchronous_Delay_Until(T, AD);
      T1.E;
   exception
   when Abort_Signal => null;
   end;
...
end T1;
```

The `AD` object provides a handle by which the application can cancel the delay with the `Cancel_Delay` operation. This would be used in implementing operations such as `Call_Timed`, where the timeout should not take effect once the associated rendezvous has started.

This scheme has the disadvantage that all operations involving time within the RTS must be dispatching, with the associated overhead in terms of execution time and runtime data structures.

5.4 A SEPARATE SOFTWARE LAYER

Our solution to this problem is to eliminate references to time in the GNARL proper. All references to time and timeouts are implemented by a layer on top of the GNARL: the "Timer Service" layer of Figure 1. So far as the GNARL is concerned, all timeout operations are the responsibility of the application. This is made possible by features introduced into Ada 9X, in particular protected objects and asynchronous transfer of control. This allows an Ada implementation the freedom to implement multiple time types without the need to modify either the GNARL interface or its implementation. An additional benefit is the ability for users to add additional time types.

The constructs for which such time parameters would be expected to appear are:

- simple delay statement (relative and absolute)
- asynchronous select with delay trigger
- timed task entry call
- timed protected entry call
- selective wait with delay alternative

The GNARL strategy is:

1. Transform simple delay statements into protected entry calls, on a "delay object" that is provided for each time type.
2. Transform all the other constructs into other Ada structures, using simple delay statements or asynchronous protected entry calls on a delay object.

The translation strategy will be explained in more detail, for each construct.

5.4.1 SIMPLE DELAY STATEMENT

For each time type, the name of the *time_package* defining the time type can be made known to the compiler, e.g. via an implementation-defined pragma. Alternatively, the compiler may be able to determine whether the argument to a **delay until** clause is a time type value by inspecting its type for the required characteristics. The time type is required to be non-limited, and to have a relational operation `"<"`; the compiler is allowed to assume that `"<"` is transitive.

The *time_package* is also required to have a child package *time_package*.`Delays` that defines a protected object `Delay_Until_Object` with an entry `Wait` with a single in-parameter of the time type. The effect of a call to the `Wait` entry should be to queue the call until the time specified by its argument. This child package is also required to define a constant `Max_Time`, which the compiler can assume is greater than all other values of the type (as reported by the `"<"` operation).

```
package Some_Time_Type.Delays is
   Max_Time : constant Time := implementation-defined;
   protected Delay_Until_Object is
      entry Wait(T: Time);
   end Delay_Until_Object;
end Some_Time_Type.Delays;
```

A simple delay statement "**delay until** `T;`" is translated to a call on the `Wait` entry associated with the appropriate time type.

```
  Some_Time_Type.Delays.Delay_Until_Object.Wait(T);
```

Note that this translation can also be used by the compiler for relative delays, on type `Duration`, as well as for absolute delays on types `Calendar.Time`, `Real_-Time.Time`, and new types defined by users. In general, there is no requirement that the entry `Wait` of a time type treat its argument in any specific way, or that the time type actually represent "time" in any conventional sense.

5.4.2 ASYNCHRONOUS SELECT WITH DELAY TRIGGER

Asynchronous delay statements are transformed to asynchronous protected entry calls to the appropriate `Wait` entry.

Consider the following asynchronous **select** statement with a **delay until** trigger:

```
select
  delay until T;
then abort
  S;
end;
```

This can be implemented by the following code fragment:

```
select
  Some_Time_Type.Delays.Delay_Until_Object.Wait(T);
then abort
  S;
end;
```

5.4.3 TIMED TASK ENTRY CALL

Timed task entry calls are transformed to asynchronous task entry calls, where the abortable part is a simple

delay or **delay until** statement. For example, the timed entry call:

```
select
  T.E;
  S1;
or delay until Calendar.Time_Of(1993, 12, 10);
  S2;
end select;
```

might be translated as:

```
declare
   Timed_Out : Boolean := True;
begin
   select
      T.E;
      Timed_Out := False;
      S1;
   or abort
      delay until Calendar.Time_Of(1993, 12, 10);
   end select;
   if Timed_Out then
     S2;
   end if;
end;
```

The translation for the timed entry call above uses a **delay until** statement, which of course would itself be implemented as an entry call on the `Wait` entry of the delay object for the appropriate time type.

5.4.4 TIMED PROTECTED ENTRY CALL

A timed protected entry call also be transformed into an asynchronous select statement. For example, consider the following code.

```
select
  PO.E1(1);
  S1;
or delay 10.0;
  S2;
end select;
```

This might be translated as

```
declare
   Timed_Out : Boolean := True;
   Deadline: System.Real_Time.Time:=
         System.Real_Time.Clock +
         System.Real_Time.To_Time_Span(10.0);
begin
   select
      T.E;
      Timed_Out := False;
      S1;
   or abort
      delay until Deadline;
   end select;
   if Timed_Out then
     S2;
   end if;
end;
```

Note that this example is more complicated because we have chosen to treat the case of a relative delay, on type `Duration`. The `Duration` value needs to be translated to an absolute time, here a value of `System.-Real_Time.Time`, since `Protected_Entry_Call` might have to wait for `Object` to be unlocked. This can take an arbitrary amount of time, and it must be done before the **delay** is started. If a relative delay were used, the actual delay would include the time required to pend the entry call, resulting in an arbitrarily late timeout. For a timed entry call using the **delay until** statement, which would be the case for all user-defined time types, this extra transformation is not necessary.

5.4.5 SELECTIVE WAIT WITH DELAY ALTERNATIVE

Select statements are transformed into calls to an RTS routine, `Selective_Wait`, from within an asynchronous select statement with a **delay** statement as trigger. The asynchronous select statement provides the timeout. Unlike the transformations above, this construct cannot be represented by using delay objects with Ada tasking constructs. The problem is that a selective wait statement cannot be the trigger of an asynchronous **select** statement. To time out a selective wait, it would be necessary to put it in the abortable part of an asynchronous **select** with a call to a delay object acting as the trigger. This would allow the delay trigger to abort the select statement once it has accepted an entry call, contrary to Ada semantics.

For this reason, it is necessary to show the transformation of the selective wait itself into calls into the GNARL. For example, the selective wait statement below

```
select
  when B1 => accept A do S1; end A; T1;
  or when B2 => accept B; T2;
  or when B3 => delay until Time1; T3;
  or when B4 => delay until Time2; T4;
  else T5;
end select;
```

would be transformed into

```
declare
   X: Select_Index:= No_Rendezvous;
   Q: Accept_List;
   P: System.Address;
   T: Some_Time_Type.Time:= Max_Time;
begin
   Q:= ((Null_Body => False,E => A),
        (Null_Body => True, E => B));
   if not B1 then Q(1) := Null_Task_Entry; end if;
   if not B2 then Q(2) := Null_Task_Entry; end if;
   -- or some "optimized" version of the above.

   -- Try for immediate rendezvous first.

   Defer_Abortion;
```

```
   Selective_Wait(Q, Else_Mode, P, X);
   begin
      Undefer_Abortion;
      case X is
      when No_Rendezvous => null;
      when 1 => S1;
      when 2 => T2;
      when others => null;
      end case;
   at end
      if X /= No_Rendezvous and then
            not Q(X).Null_Body then
         Exceptional_Complete_Rendezvous(
               Current_Exception);
      end if;
   end;

   -- If immediate rendezvous does not work, try
   -- waiting for the specified delay.

   if X = No_Rendezvous then

      if B3 and then Time1 < T then T := Time1; end if;
      if B4 and then Time2 < T then T := Time2; end if;
      -- or some "optimized" version of the above.

      select
         Some_Time_Type.Delays.Delay_Object.Wait(T);
      then abort
         Defer_Abortion;
         Selective_Wait(Q, Simple_Mode, P, X);
      end select;

      if X = No_Rendezvous then
         Undefer_Abortion;
         if Time1 > Time2 then
            T4;
         else
            T3;
         end if;
      else
         begin
            Undefer_Abortion;
            case X is
            when No_Rendezvous => null;
            when 1 => S1;
            when 2 => T2;
            when others => null;
            end case;
         at end
            if not Q(X).Null_Body then
               Exceptional_Complete_Rendezvous(
                     Current_Exception);
            end if;
         end;
      end if;

   end if;
end;
```

The `Selective_Wait` call implements the various forms of selective wait; further details are available the GNARL Interface document [5]. Notice that `Selective_Wait` is first called with `Else_Mode`, and only if this fails is the asynchronous select statement entered. This two-stage transformation is necessary to allow the rendezvous to take place immediately, when that is possible. Otherwise, a very short timeout value might cancel the `Selective_Wait` before it is able to accept a call that was already queued, contrary to Ada semantics.

The sequence of statements preceded by **at end** is pseudo-code syntax for a transformation the compiler needs to perform on the object code, which we cannot express in Ada 9X syntax. This always precedes the **end** of a master. The compiler is responsible for ensuring that the sequence of statements between the **at end** and the **end** of the master will be executed whenever that master (in the Ada sense, including **accept** statements) is left. The `Defer_Abortion` call is used to prevent abortion from taking place once the rendezvous has started until the select alternatives are protected by an **at end** handler assuring that the end of the rendezvous will be recorded. Note that `Selective_Wait` will return if it is aborted while suspended waiting for a rendezvous. In this case, the abortion will take effect when `Undefer_Abortion` is called.

As an illustration of the meaning of the **at end** handler, consider the following two code fragments:

```
declare                  declare
  D1;                      D1;
begin                    begin
  S1;                      S1;
at end                     S2;
  S2;                    exception
end;                     when others => S2;
                         end;
```

Their effect is similar except that

- `S2` is also executed when an exception is propagated from the declarative part of the master, and
- `S2` is also executed when the task is aborted while executing anywhere in the declarative part or body of the master;
- abortion is deferred before control is transferred to `S2`, and it is undeferred before control leaves the master (unless the entire construct is contained in another abort-deferred region).

6 CONCLUSIONS

The GNARL provides tasking support that meets the design goals of portability, efficiency, interoperability with other languages, and user extensibility by being written in Ada 9X, through its layered design, and through its use of a standard interface for multithreaded services.

ACKNOWLEDGEMENTS

The technical contributions of the GNAT development team are gratefully acknowledged. The ideas expressed regarding the elaboration of Ada from other languages owe much to discussions between members of the Ada 9X MRT mail group (ada9x-mrt@inmet.com).

References

[1] Ada 9X Mapping/Revision Team. *Ada 9X Reference Manual: Draft Version 4.0*, September 1993. Available by anonymous FTP from ajpo.sei.cmu.edu.

[2] Ted Baker. Requirements specification for the Common Ada Run-Time System. Prepared for Westinghouse Electric Corporation, February 1991.

[3] T.P. Baker and E.W. Giering. Gnu low-level interface definition. Technical report, Florida State University, June 1993. Available by anonymous FTP from ftp.cs.fsu.edu.

[4] T.P. Baker and E.W. Giering. Gnu low-level interface definition annex: Condition variables. Technical report, Florida State University, June 1993. Available by anonymous FTP from ftp.cs.fsu.edu.

[5] T.P. Baker and E.W. Giering. PART/GNARL interface definition. Technical report, Florida State University, April 1993. Available by anonymous FTP from ftp.cs.fsu.edu.

[6] E.W. Giering and T.P. Baker. Using POSIX threads to implement Ada tasking: Description of work in progress. In *TRI-Ada '92 Proceedings*, pages 518–529. ACM, November 1992.

[7] E.W. Giering, Frank Mueller, and T.P. Baker. Implementing Ada 9x features using POSIX threads: Design issues. In *TRI-Ada '93 Proceedings*, pages 214–228. ACM, September 1993.

[8] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1986.

[9] SunSoft. *SunOS 5.3 Linker and Libraries Manual*, 1993.

[10] New York University. The Gny NYU Ada Translator (GNAT). Available by anonymous FTP from cs.nyu.edu.