Ada 9x Asynchronous Transfer Of Control: Applications And Implementation *

E.W. Giering III, T.P. Baker†
Department of Computer Science
Florida State University
Tallahassee, FL 32306-4019
Internet: giering@cs.fsu.edu Phone: (904)644-3441

Abstract

The draft revised Ada language standard provides several new constructs which support the so-called imprecise computation technique. That is, an algorithm in progress can be interrupted in response to a real-time event, and a partial result can be recovered. This new capability, called Asynchronous Transfer of Control, allows a dynamic tradeoff between accuracy and timeliness, which is essential in some real-time software systems. It also provides interesting challenges for language implementors.

1 Introduction

The correctness of hard real-time programs is dependent on their timing; they are considered to have failed if they do not meet their timing specifications. Real-time programs are also subject to other measurements of correctness, in particular the accuracy of their computations. Since some computations generate results of increasing accuracy the longer they run, real-time programs can be designed to perform tradeoffs between accuracy and time, depending on which is more critical [11]. When this can be done dynamically during scheduling, real-time programs can be made to degrade gracefully, providing acceptable answers in a timely fashion rather than accurate answers when they are no longer useful.

To allow for the dynamic tradeoff between accuracy and speed, it must be possible to interrupt an algorithm in progress and recover its best result. A new language feature proposed by the emerging revised Ada language standard (Ada 9X) [1] provides this capability in the form of Asynchronous Transfer of Control (ATC), a mechanism which allows an event to interrupt a computation in progress. This paper begins by illustrating the use of ATC in coding imprecise computations. It then proceeds to discuss implementation issues.

*Published in Proceedings of the ACM SIGPLAN Workshop on Language, Compiler, and Tool Support for Real-Time Systems, June 21, 1994.

†Partially funded by the Ada Joint Program Office under the Ada Technology Insertion Program, through HQ U.S. Army CECOM.

2 Application of ATC

As an example of an algorithm that iteratively improves its results, we will use the following:

```
generic
  function F(X : Float) return Float;
procedure Zero
  (Tolerance : Float;
   High, Low : in out Float);
procedure Zero(...) is
  Midpoint : Float;
begin
  if F(High) * F(Low) > 0.0 then raise No_Zero; end if;
  while abs(High - Low) > Tolerance loop
    Midpoint := (High + Low) / 2.0;
    if F(Midpoint) * F(Low) > 0.0 then
      Low := Midpoint;
    else High := Midpoint;
    end if;
  end loop;
end;
```

This procedure performs a binary search for a zero of the generic function `F` in the interval [`High`, `Low`]. It returns its results as a new interval [`High`, `Low`] which is no wider than `Tolerance`, but which still contains a zero. It checks that the function value at the two endpoints are of different sign, and raises the exception `No_Zero` if this is not true.

This routine will execute until the result is within the requested tolerance, regardless of timing constraints. One way to make it sensitive to timing constraints is to implement a polling scheme, where the routine checks periodically whether it has exceeded its deadline, e.g

```
...
procedure Zero_Polled(...
    Deadline : System.Real_Time.Time;...) is
...
begin ...
  while System.Real_Time.Clock < Deadline and then
      abs(High - Low) > Tolerance loop
  ...
  end loop;
  ...
end;
```

A problem with this approach is the latency between the routine's exceeding its deadline and returning to the caller. In the example, this could be as long as one whole iteration of the algorithm, which depends on the complexity of the function `F`. For complex algorithms it may be difficult to find polling points that make the latency predictable. Even where it can be done, inserting polling code within an algorithm breaks down a desirable separation of concerns between the detailed coding of the algorithm and system-level scheduling. Instead of polling, it would be preferable to be able to interrupt the task [1] performing the imprecise computation immediately when it has reached the specified deadline, with negligible (or at least predictable) latency.

The only way to implement an interruptible operation in Ada, prior to Ada 9X, is to create a separate *agent* task to execute the monotonic computation. When the deadline arrives, the agent task can be aborted. The agent task must write its intermediate results to a global location where it can be read by the client, or communicate the results back by rendezvous[2].

In Ada 83, it is difficult for the agent to communicate back the partial results, without risk of the data being corrupted if the agent is aborted during the communication. One way to do this is to use a rendezvous to update the intermediate results, allowing the client task to use rendezvous to wait until no update is in progress before aborting the agent task. This requires another task to service the update entry call.

```
...
procedure Zero_With_Deadline(...) is
  task Result is
    entry Put_Result(Hi, Lo : Float);
    entry Quit;
  end Result;
  task Agent is
    entry Done;
  end Agent;
  task body Result is
    Done : Boolean := False;
  begin
    while not Done loop
      select
        accept Put_Result(Hi, Lo : Float) is
          High := Hi;
          Low := Lo;
        end Put_Result;
      or accept Quit; Done := True;
      end select;
    end loop;
  end Result;
  task body Agent is
    Midpoint : Float;
    Hi : Float := High;
    Lo : Float := Low;
  begin
    if F(Hi) * F(Lo) > 0.0 then raise No_Zero; end if;
    while abs(Hi - Lo) > Tolerance loop
      ...
      Result.Put_Result(Hi, Lo);
    end loop;
    accept Done;
  end Agent;
begin
  select Agent.Done;
  or delay Deadline - Calendar.Clock;
  end select;
  Result.Quit;
  abort Agent;
end Zero_With_Deadline;
```

The procedure above creates local tasks `Agent`, to perform the required computation, and `Result`, to synchronize the intermediate results. The agent task calls the `Put_Result` in `Results` to move its internal intermediate results to the parameters of `Zero_With_Deadline` for return.

The deadline is enforced by the task that calls `Zero_With_Deadline` (the client). The client uses a *timed entry call* (the **select** construct) to wait for either the agent task to finish and execute its **accept** statement or the deadline to arrive. When this occurs, the client performs a rendezvous with the synchronization task. The **select** statement in `Result` allows either `Put_Result` or `Quit` to rendezvous, but only one of them can rendezvous at a time. The rendezvous with `Quit` therefore assures that the the client's call to `Put_Result`, if any, is complete before interrupting the computation, and also causes the synchronization task to terminate, preventing any further update of the intermediate results. Note that this can only delay the client for the amount of time needed to do the update, a short and predictable latency to aborting the computation. Once the client is sure that the intermediate results are consistent, it aborts the agent task, which stops it from executing if it has not already done so.

One problem, prior to Ada 9X, is that task abortion was not required to be preemptive. An implementation was allowed to defer the abortion of a task until the next time it executed one of a specified set of operations. The aborted task may continue executing forever, if it never executes one of these designated abortion "synchronization points". Effectively, this reduces to polling, and suffers from the same defects. This problem is addressed by the Ada 9X Real-Time Systems annex, which requires that abortion be immediate and preemptive, outside of those special contexts where it is required to be deferred.

Another problem with this approach is that the overhead cost for creation and termination of a separate task for the agent is likely to be high [6, 7]. Therefore, the use of an agent task is only likely to be acceptable if the computation performed by the agent is long enough that this overhead is comparatively insignificant.

To address this problem, Ada 9X introduces the asynchronous **select** statement. This permits selective abortion of a sequence of statements within a task, in response to an arbitrary asynchronous external event, without aborting the entire task. Ada is the first language proposed for international standardization to provide such a construct, though several languages have previously provided features limiting statement-level constructs to specific amounts of real or execution time [3, 5, 9, 10].

The syntax of the asynchronous **select** statement is

```
select
   triggering_statement [ triggered_statements ]
then abort
   abortable_part
end select;
```

The triggering statement specifies the event that will abort the abortable part. It can be either an entry call or a delay statement. Either of these could ordinarily

[1] A *task* is an Ada object with an independent thread of control.

[2] *Rendezvous* is an operation in which two tasks synchronize and may exchange data. A rendezvous takes place when one task has called an *entry* of the another task, and the owner of the entry executes an *accept* statement for that entry.

cause the task to block, but in the context of an asynchronous **select** statement, instead of blocking it will execute the sequence of statements in the abortable part.

When the triggering statement completes, the execution of the abortable part is aborted, and control is transferred to the triggered statements. This is the asynchronous transfer of control. If the abortable part completes before the triggering statement starts, the triggering statement is cancelled and control is transferred to the continuation of the asynchronous **select** statement. In either case, if the abortable part has dependent tasks or controlled objects[3], finalization actions must be performed before exiting it.

Ada 9X provides several new tools to prevent abortion of a computation from corrupting intermediate results. One is the pragma `Volatile`, which instructs the compiler to implement every logical update or reference to the value of a particular variable directly to (shared) memory. This permits use of a write-twice/read-twice scheme to ensure there is always one uncorrupted copy of the partial result, with no further need for synchronization. If the result is sufficiently small that hardware-supported atomic load and store operations are adequate, redundant writing and reading can be eliminated, by using the pragma `Atomic` which instructs the compiler to use such atomic operations exclusively for a particular variable.

A more general mechanism for reporting partial results, atomically with respect to abortion, is to use a protected object. Protected objects are a new Ada 9X feature, similar the classic Hoare monitor. An object of a protected type has hidden data and exports a set of operations (subprograms, and entries) on that data. Unlike a task, it does not have its own thread of control. The language implementation is required to enforce mutual exclusion between the exported operations, and defer the abortion of a task while it is executing such an operation.

In the following version of `Zero_With_Deadline`, an absolute delay statement[4] is used as the triggering statement for an asynchronous **select** statement, enabling the agent task to be eliminated. The results are synchronized using the protected object `Result`, allowing the data synchronization task to be eliminated as well. Finally, we show the use of an Ada 9X subprogram access parameter to provide the function `F`, eliminating the need to instantiate a generic for each function to be analyzed.

```
type Func is access function(X : Float) return Float;
procedure Zero_With_Deadline(...F : Func...) is
  protected Result is
    -- operations cannot be aborted
    procedure Read(High, Low : out Float);
    procedure Write(High, Low : Float);
  private
    Hi : Float := High;
    Lo : Float := Low;
  end Result;
  protected body Result is
    procedure Read(High, Low : out Float) is
    begin High := Hi; Low := Lo;
    end Read;
    procedure Write(High, Low : Float) is
    begin Hi := High; Lo := Low;
    end Write;
  end Result;
begin
  if F(High) * F(Low) > 0.0 then
    raise No_Zero;
  else
    select delay until Deadline;
    then abort
      while abs(High - Low) > Tolerance loop
        ...
        Result.Write(High, Low);
      end loop;
    end select;
  end if;
  Result.Read(High, Low);
end;
```

If the deadline arrives before the final write is performed, ATC terminates the loop after waiting for any pending calls to `Result.Write` to complete and transfers control to the call to `Result.Read`.

3 Implementing ATC

This section describes the implementation problems presented by ATC, some techniques that may be used to implement it, and some of the tradeoffs involved in choosing between them. Further discussion of this topic can be found in [7, 8].

An implementation of ATC must address the following issues:

- interruption of the target task, and initiation of abortion
- deferral of abort over certain regions
- execution of finalization procedures for any local objects in the aborted part, each in its correct context
- finding the proper location and context to continue execution, after the ATC
- handling nested scopes, including nested asynchronous **select** statements
- ensuring safety of compiler-generated code sequences, including subprogram call and return, when interrupted by ATC.

Note that these issues are not unique to the asynchronous **select** statement. Most of them also come up with abort of an entire task, so long as the effect of the abort must be immediate and the aborted task must execute finalization code between the abort and termination.

Since ATC is not likely to be used in most (non-real-time) Ada programs, a key objective of any implementation should be to impose little or no distributed overhead for the existence of this feature. Subject to this constraint, the implementation of ATC itself should be as efficient as possible. To justify its special syntactic

[3] *Controlled objects*, introduced in Ada 9X, inherit initialization and finalization operations.

[4] Absolute delay statements are new feature in Ada 9X

status, it should at least be more efficient than if the user explicitly created and aborted an agent task, as in our earlier example.

3.1 One- vs. Two-Thread Models

The canonical implementation model, after which ATC is named, is to asynchronously transfer control within a task—i.e. to reset the instruction counter to a value outside the normal flow of control [12]. To execute an asynchronous **select** statement, the task starts with the triggering statement. If it can be completed immediately, the abortable part is never executed at all; the task continues with the triggered statements, and then jumps to the end of the **select** statement. Otherwise, the task enqueues itself as "waiting" for the event required to complete the triggering statement, but does not suspend execution; instead it proceeds to execute the abortable part. If triggering event occurs before the task completes the abortable part, the task is interrupted and forced to first execute finalization code for the abortable part, then transfer control to the triggered statements. Otherwise, if the task completes the abortable part first, it dequeues itself, and then jumps to the end of the **select** statement. This is the *one-thread model* of ATC.

Some Ada implementors have proposed a *two-thread* implementation model. This involves translation of the asynchronous **select** statement into code similar to the agent-task example above. Before blocking on the triggering statement, the task executing the asynchronous `select` creates an agent thread of control to execute the abortable part. The ATC is done by aborting the agent thread, if the original thread wakes up at the triggering statement before the agent completes.

Proponents of the two-thread model have argued that it simplifies the implementation in several respects. One is that it preserves two useful invariants of the pre-Ada 9X tasking model namely: (1) a thread that is waiting for an event is not executing; (2) a thread never waits for more than a timeout and one other event. Another simplification is that the two-thread model eliminates the need for one thread to asynchronously modify another thread's flow of control, which is not possible in some execution environments. If there is a way to kill a thread, it should be sufficient to simply kill the agent thread and wake up the client.

The two-thread model seems to complicate the implementation at least as much as it simplifies it. It violates a key invariant of Ada tasking, that there is a one-to-one correspondence between tasks and threads of control. This assumption pervades the semantics, and is the foundation of existing Ada tasking implementations. Loss of this invariant has many ramifications. Among these, data that previously could only be accessed by one thread of control becomes susceptible to race conditions. Thus, there are new requirements for synchronization, and new potential for deadlock within a single task. Also, just killing the agent thread is not such a simple solution as it might seem. There remains the problem of how to execute the agent's finalization code. If the operation that kills a thread does not support finalization, some other thread must perform the finalization. To do so, it must wait for the killed thread to be terminated, and then obtain access to the runtime stack of the terminated thread. The latter may not be possible in systems where killing a thread also releases its stack space.

Detailed RTS complexity arguments aside, the implicit creation and destruction of an agent thread of control seems bound to incur most of the overhead as if the user had coded the ATC using an explicit agent task, thereby nullifying the main objective of the asynchronous **select** statement.

For these reasons, we will assume in the rest of this paper that ATC is implemented according to the one-thread model.

3.2 Interrupting the Target Task

If the effect of abort is to be immediate, the execution of the target task must be interrupted. How difficult this is depends on the execution environment. On a single processor, and assuming the Ada runtime system (RTS) has direct control of the hardware, it is simple. The task initiating the abort must be executing on the processor. Unless it is trying to abort itself, it must already have preempted the target task. In this case, it is sufficient to modify the saved state of the target task to redirect control; the target task will begin processing the abort as soon as it is next scheduled to execute. If the task is aborting itself, and abort is not deferred (see below), it can begin processing the abort immediately.

On a multiprocessor, it is possible that the target task is executing on a different processor than the task that initiates the abort. If the architecture allows one processor to interrupt another, and the Ada RTS has direct control of the hardware, it can interrupt the target task's processor. The interrupt handler can then modify the saved state of the target task, causing it to begin processing the abort on return from the interrupt. If the architecture does not permit direct interruption, it may be necessary to delay the abortion until the next clock interrupt, or the next time the target task calls the RTS, whichever comes first.

Things are not so simple when the Ada RTS is layered over a more general-purpose operating system, and Ada tasks are implemented as separate OS processes/threads. Since the OS implements process/thread scheduling and context switching, the Ada RTS may not be able to modify the saved context of a task, even on a single processor. On a multiprocessor, it may not be able to interrupt the execution of an executing task. A friendly OS may provide a mechanism that can be adapted. For example, the proposed IEEE standard POSIX threads interface provides a per-thread signaling mechanism, which can be used to force the target thread to execute a handler. In a less friendly environment, it may be necessary to delay processing the abort until the next time (if any) the target task calls the Ada RTS, or if more immediate response is required, the compiler may need to insert polling code in positions that are guaranteed to break every loop.

3.3 Deferring Abort

There are critical regions of code were the processing of an abort must be deferred. For example, abort deferral is required by the language for protected actions and finalization routines. It is generally also required during the execution of the Ada RTS, to ensure the integrity of RTS data structures.

Implementing abort deferral can be divided into two parts:

- Determining whether abort is deferred for a given task, at the point it is targeted for abortion.
- Ensuring deferred aborts are processed immediately when abort-deferral is lifted.

In general, the determination of whether a given task is abort-deferred must be done by the task itself. In a single-processor system, it may be possible for the task initiating an abort to determine whether the target task is abort-deferred. However, in a multi-processor system, or a single processor system where the Ada RTS is not in direct control of task scheduling, this is not possible. The abort-deferral state of the target task may change between the point it is tested and the point the target task is interrupted.

There are two obvious techniques for recording whether a task is abort-deferred. One is for the task to increment and decrement a deferral nesting level (e.g. in a dedicated register or the task control block), whenever it enters and exits an abort-deferred region. On exit from such a region, if the counter goes to zero, the task must check whether there is a pending abort, and process the abort if so. This deferral-counter method imposes some distributed overhead on entry and exit of abort-deferred regions, but allows quick checking.

The other technique is sometimes termed PC-mapping. The compiler and link-editor generate a map of abort-deferred regions. Whether the task is abort-deferred can then be determined by comparing the current instruction-pointer value, and all the saved return addresses of active subprogram calls, against the map. To ensure the abort is processed on exit from the abort-deferred region, one overwrites the saved return-address of the outermost abort-deferred call frame with the address of the abort-processing routine (saving the old return address elsewhere). The test for abort-deferral may take time proportional to the depth of subprogram call nesting, but that occurs only if an ATC is attempted. Until that occurs, no runtime overhead is incurred for abort deferral. A restriction of this method is that abort-deferred regions must correspond to callable units of code. Another restriction is that the subprogram calling convention is constrained, to (1) ensure the return addresses are always in a predictable and accessible location, and (2) ensure this data is always valid, even if the calling sequence is interrupted. Unfortunately, that is not true for some architectures.

3.4 Processing the Abort

In general, processing an abort requires unwinding the stack of the target task, rather than immediately jumping out of the aborted part (or killing the task, in the case of entire-task abortion). There may be local controlled objects, which require the execution of a finalization routine. There also may be dependent tasks, which require the abort processing block until they have been aborted, finalized, and terminated. The finalization must be done in LIFO order, and the stack contexts of the objects requiring finalization must be preserved until after the objects are finalized. In this respect, ATC is very much like exception propagation[5], so it is desirable that one mechanism serve for both purposes.

In principle, some efficiency might be gained by avoiding detailed unwinding of the stack, executing the finalization routines from the top of the stack or from a different stack, then popping the entire stack down to the context where control is to be transferred. However, this presumes there is some way to recover that context without full unwinding. If the compiler uses a callee-save register spilling convention, there may be values of live registers spilled at unpredictable locations on the stack. In this case, it seems one must create a register save area for each potential target of ATC (analogous to jump-buffer implementation of C's *setjmp()* and *longjmp()* operations). While asynchronous **select** statements may not be very common, exception handlers are common (some implicitly provided by the compiler), and controlled objects are also expected to be common. Thus, the overhead of creating a jump-buffer for every potential asynchronous transfer point is objectionable.

Some means must be provided for locating finalization routines, and the point at which execution is to resume after an ATC. This problem is very similar to that of finding an exception handler [2], and the same solutions apply. The main approaches are saving a pointer in the stack frame for each scope, PC-mapping, and various hybrids of the two. The PC-mapping approach is generally preferable, since it imposes no distributed overhead on execution. If PC-mapping is used for the latter purpose, there is strong motivation to try to make the same technique serve double duty, for

[5] *Exceptions* are detected errors which cause transfer of control to a matching handler, in a dynamically enclosing construct.

abort-deferral.

4 Conclusions

Ada 9X includes several new features which are intended to support real-time programming using partial results. Central among these are asynchronous transfer of control, provided by the asynchronous **select** statement, and the deferral of abortion over certain critical sections of code. These new features poses some interesting implementation challenges, which involve not only the Ada runtime system but also interactions with the code generation done by the compiler. Although implementation techniques are known, further work is needed to determine how to achieve the best performance with these new features. For example, it would be interesting to find out whether the indirect overhead of using calling sequences that permit a PC-mapping implementation of ATC is greater than the overhead of maintaining and checking a deferral-nesting counter.

The authors are currently implementing the GNu Ada 9X Runtime Library (GNARL), including support for real-time tasking, for the Gnu NYU Ada Translator (GNAT) [13]. This work is in cooperation with the GNAT development team at New York University. It is hoped that the public availability of this code as a starting point should enable more experimentation with Ada implementation techniques than has been possible heretofore.

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] T.P. Baker and G.A. Riccardi. Implementing Ada exceptions. *IEEE Software*, pages 42–51, September 1986.

[3] Gérard Berry, Sabine Moisan, and Jean-Paul Rigault. ESTEREL: Towards a synchronous and semantically sound high level language for real time applications. In *Proceedings of the Real-Time Systems Symposium*, Arlington, Virginia, December 1983.

[4] Jen-Yau Chung, Jane W. S. Liu, and Kwei-Jay Lin. Scheduling periodic jobs using imprecise results. Technical Report UIUCDCS-R-87-1307, Departement of Computer Science, University of Illinois, November 1987.

[5] Narain Gehani and Krithi Ramamritham. Real-time concurrent C: A language for programming dynamic real-time systems. *Real-Time Systems*, 3(4):377–405, December 1991.

[6] E.W. Giering and T.P. Baker. POSIX/Ada real-time bindings: Description of work in progress. In *Proceedings of the Ninth Annual Washington Ada Symposium*. ACM, July 1992.

[7] 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.

[8] 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.

[9] Insup Lee and Vijay Gehlot. Language constructs for distributed real-time programming. In *Proceedings of the Real-Time Systems Symposium*, San Diego, California, December 1985.

[10] Kwei-Jay Lin and Swaminathan Natarajan. Expressing and maintaining timing constraints in FLEX. In *Proceedings of the Real-Time Systems Symposium*, Huntsville, Alabama, December 1988.

[11] Jane W. S. Liu, Kwei-Jay Lin, and Swaminathan Natarajan. Scheduling real-time periodic jobs using imprecise results. In *Proceedings of the Real-Time Systems Symposium*, San Jose, California, December 1987.

[12] Ada 9X Mapping/Revision Team. Ada 9X implementation modules II–IV. Technical report, Intermetrics, Inc., June 1991. Available by anonymous FTP from ajpo.sei.cmu.edu.

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