A Comparison of the Asynchronous Transfer of Control Features in Ada and the Real-Time Specification for Java™

Benjamin M. Brosgol
Ada Core Technologies
79 Tobey Road, Belmont MA 02478, USA
brosgol@gnat.com

Andy Wellings
Dept. of Computer Science, University of York
Heslington, York YO10 5DD, UK
andy@cs.york.ac.uk

11 February 2003

Abstract

Asynchronous Transfer of Control ("ATC") is a transfer of control within a thread,[1] triggered not by the thread itself but rather from some external source such as another thread or an interrupt handler. ATC is useful for several purposes; e.g. expressing common idioms such as timeouts and thread termination, and reducing the latency for responses to events. However, ATC presents significant issues semantically, methodologically, and implementationally. This paper describes the approaches to ATC taken by Ada [1] and the Real-Time Specification for Java [2, 3], and compares them with respect to safety, programming style / expressive power, and implementability / latency / efficiency.

Overview

Section 1 introduces the basic terminology and sets the framework for the discussion of ATC. Section 2 presents the fundamental issues surrounding ATC and briefly summarizes the Ada and Real-Time Specification for Java ("RTSJ") approaches. Section 3 describes and analyzes Ada's approach to ATC. Section 4 summarizes and critiques the ATC facilities in regular Java. Section 5 gives a general overview of the RTSJ, and Section 6 then describes and analyzes the RTSJ's approach to ATC. Section 7 contrasts the Ada and RTSJ approaches. Section 8 summarizes this paper's findings. Appendix A presents a complete ATC example with sample Ada and RTSJ solutions and compares the two versions.

This paper is an extended version of [4].

1 Preliminaries

Communication between threads may be either *synchronous* or *asynchronous*. In the synchronous case, the receiving thread waits for a communication request from a sender. Examples of synchronous communication are the Ada rendezvous, the Java `wait` / `notify` mechanism, and the setting and polling of shared data. In the asynchronous case, the receiving thread is informed "immediately" of the sending thread's request to communicate; the receiver does not need to be waiting for the request. There are

[1] We use the term "thread" generically to refer to a concurrent activity within a program. When discussing a particular language's mechanism we use that language's terminology (e.g., "task" in Ada).

two models for such asynchronous communication: one is based on *resumption*, and the other on *termination*. The models are not mutually exclusive; a multithreading language or operating system may support either or both.

With the resumption model, a communication request results in interruption of the receiver's thread of control, and execution of a "handler" that is supplied as part of the receiver's logic. The handler runs on the receiver's stack and consumes the receiver's execution time budget. When the handler finishes, the receiver continues executing from the point where it was interrupted.

Most operating systems provide this kind of facility through signals and signal handling, if not at the thread level then at the process level. For example, in POSIX [5] a signal `sig` is sent to a particular pthread `pt` if the system can determine that `pt` is the pthread that generated `sig`, but `sig` is sent to the process as a whole if the generating pthread cannot be identified.

With the termination model the receiver's thread of control is similarly interrupted, and a handler is executed. However, control does not resume at the point of interruption. Instead, execution of the context in which the receiver was executing is abandoned (either immediately or upon exit from a region where such interruption is deferred), a handler (if one exists) associated with that context is invoked, finalization code (if any) is executed, and the context is exited. In this paper, the term *Asynchronous Transfer of Control* (or simply *ATC*) is used to denote asynchronous communication via such a termination model.

Direct operating system support for ATC is generally fairly limited, typically comprising mechanisms for thread termination versus more fine-grained control. For example, POSIX supports a pthread cancellation facility through which a pthread is allow to push and pop "cleanup" handlers as it enters and leaves execution contexts. When a pthread is canceled, its contexts are popped and the associated handlers are executed. When all handlers have run, the pthread terminates. POSIX also allows the pthread to defer cancellation during sensitive code.

Ada and the Real-Time Specification for Java support both the resumption and termination models for asynchronous communication. This paper will focus on the latter.

2 Summary of Issues and Approaches

ATC is a rather controversial feature. It is methodologically suspect, since writing correct code is difficult if control transfers can occur at unpredictable points. As an example, if a thread has acquired a shared resource such as a mutex lock and then incurs an ATC before releasing the resource, the result may be deadlock or "resource leakage". Moreover, ATC is complicated to specify and to implement, and in the absence of optimizations it may incur a performance penalty even for programs that don't use it.

Despite these difficulties, there are several situations that are common in real-time applications where ATC offers a solution:

Timing out on a computation. A typical example is a function that performs an iterative calculation where an intermediate approximation after a given amount of time is required.

Terminating a thread. An example is a fault-tolerant application where because of a hardware fault or other error a thread might not be able to complete its assigned work. In some situations the termination must be "immediate"; in other cases it should be deferred until after the doomed thread has had the opportunity to execute some "last wishes" code.

Terminating one iteration of a loop. The ability to abort one iteration but continue with the next is useful in an interactive command processor that reads and executes commands sequentially, where the user can abort the current command.

It is possible, and generally preferable stylistically, to program such applications via "polling" (the thread can check synchronously for "requests" to perform the given action, and take the appropriate action). However such a solution may introduce unwanted latency and/or unpredictability of response time, a problem that is exacerbated if the thread can block.

The basic problem, then, is how to resolve the essential conflict between the desire to perform ATC "immediately" and the need to ensure that certain sections of code are executed to completion[2] and that relevant finalization is performed.

[2]Note that "executed to completion" does not imply non-preemptability.

In brief, ATC in Ada is based on the concept of aborting either a task (via an explicit `abort` statement) or a syntactically-distinguished sequence of statements (as the result of the completion of a triggering action, such as a timeout, in an asynchronous select statement). The fundamental principle is that ATC should be safe: it is postponed while control is in an *abort-deferred* operation (e.g., a rendezvous), and when it takes place any needed finalizations are performed. ATC does not entail raising or handling exceptions.

In contrast, the Real-Time Specification for Java captures ATC via asynchronous exceptions, which are instances of the `AsynchronouslyInterruptedException` (abbreviated "`AIE`") class. An ATC request is posted to a target thread by a method call rather than a specialized statement, making an `AIE` instance *pending* on the thread. In the interest of safety, several contexts are *ATC-deferred*: synchronized code, and methods and constructors lacking a `throws AIE` clause. An asynchronous exception is only thrown when the target thread is executing code that is not ATC-deferred, and special rules dictate how it is propagated / handled. Finalization can be arranged via appropriately placed explicit `finally` clauses.[3] The ATC mechanism can be used for thread termination, timeout, and other purposes.

The remainder of this paper will elaborate on these points. The focus will be on uniprocessor systems; multiprocessors present some issues that are beyond the scope of this paper (and in any event the RTSJ does not explicitly address multiprocessor environments).

3 ATC in Ada

This section summarizes, illustrates, and critiques Ada's[4] ATC features.

3.1 Semantics

One of Ada's ATC constructs is the `abort` statement, which is intended to trigger the termination of one or more target tasks. However, termination will not necessarily be immediate, and in some situations (although admittedly anomalous in a real-time program) it will not occur at all. An aborted task becomes "abnormal" but continues execution as long as it is in an *abort-deferred operation* (e.g., a protected action). When control is outside such a region the aborted task becomes "completed". Its local controlled objects are finalized, and its dependent tasks are aborted. The task terminates after all its dependent tasks have terminated. (This description was necessarily a simplification; full semantics are in [1], Sections 9.8 and D.6).

Ada's second ATC feature — the *asynchronous select* statement — offers finer-grained ATC. This syntactic construct consists of two branches:

- A *triggering alternative*, comprising a sequence of statements headed by a *triggering statement* that can be either a delay statement (for a timeout) or an entry call;
- The *abortable part*, which is the sequence of statements subject to ATC.

Conceptually, the abortable part is like a task that is aborted if/when the triggering statement completes.[5] In such an event, control passes to the statement following the triggering statement. Otherwise (i.e., if the abortable part completes first) the triggering statement is canceled and control resumes at the point following the asynchronous select statement.

The following example illustrates a typical use for an asynchronous select statement. The context is a `Sensor` task that updates `Position` values at intervals no shorter than 1 second. It times out after 10 seconds and then displays a termination message. `Position` is a protected object declared elsewhere, with an `Update` procedure that modifies its value.

[3] Java also supplies finalization semantics via the `finalize()` method from class `Object`, but when (if ever) `finalize` is invoked depends on the implementation's Garbage Collection strategy. It is thus inadvisable to rely on the `finalize` method to perform needed finalizations.

[4] It is assumed that the implementation complies with the *Real-Time Systems Annex*.

[5] It is thus subject to the rules for abort-deferred operations, and finalization of controlled objects.

```
task body Sensor is
  Time_Out       : constant Duration := 10.0;
  Sleep_Interval : constant Duration := 1.0;
begin
  select
    delay Time_Out;
    Put_Line("Sensor terminating");
  then abort
    loop
      Position.Update;
      delay Sleep_Interval;
    end loop;
  end select;
end Sensor;
```

The asynchronous select statement deals with nested ATCs correctly. For example, if the delay for an outer triggering statement expires while an inner delay is pending, the inner delay will be canceled and an ATC will be promulgated out of the inner abortable part (subject to the semantics for abort-deferred operations) and then out of the outer abortable part.

The Ada ATC facility does not rely on asynchronous exceptions. It thereby avoids several difficult semantic issues; these will be described below in Section 6.1, in conjunction with the RTSJ's approach to ATC.

3.2 Safety

The Ada rules for abort-deferred operations give precedence to safety over immediacy: e.g., execution of `Update` in the `Sensor` task will be completed even if the timeout occurs while the call is in progress. Abort-deferred regions include other constructs (e.g. finalization) that logically must be executed to completion. However, the possibility for ATC must be taken into account by the programmer in order to ensure that the program performs correctly. For example, the idiom of locking a semaphore, performing a "block assignment" of a large data structure, and then unlocking the semaphore, is susceptible to data structure corruption and/or deadlock if an ATC occurs while the assignment is in progress.

Ada's two ATC features take different approaches to the issue of whether the permission for ATC is implicit or explicit. Any task is susceptible to being aborted; thus the programmer needs to program specially so that regions that logically should be executed to completion are coded as abort-deferred constructs. In any event the `abort` statement is intended for specialized circumstances: to initiate the termination of either the entire partition or a collection of tasks that are no longer needed (e.g., in a "mode change").

The asynchronous select statement takes the opposite approach. The abortable part of such a statement is syntactically distinguished, making clear the scope of the effect of the triggering condition. However, since subprograms called from the abortable part are subject to being aborted, the author of the asynchronous select construct needs to understand and anticipate such effects.

Some Ada implementations provide facilities for specifying local code regions that are abort deferred (e.g., pragma `Abort_Defer` in GNAT).

3.3 Style and Expressiveness

The Ada ATC mechanisms are concise and syntactically distinguished, making their effects clear and readable. For example, the timeout of the `Sensor` thread is captured succinctly in the asynchronous select statement.

While a task is performing an abort-deferred operation, an attempt to execute an asynchronous select statement is a bounded error. Thus a programmer implementing an abort-deferred operation needs to know the implementation of any subprograms that the operation calls.

Ada does not have a construct that immediately/unconditionally terminates a task. This omission is a good thing. Such a feature would have obvious reliability problems; e.g., if it were to take place during a protected operation, shared data might be left in an inconsistent state.

Several gaps in functionality should be noted:

Triggering "accept" statement. Ada allows an entry call but not an `accept` statement as a triggering statement. This restriction can lead to some stylistic clumsiness and the need for additional intermediate tasks.

Cleaner finalization syntax. Capturing finalization by declaring a controlled type and overriding `Finalize` can be somewhat awkward. An explicit control structure (e.g. something like Java's `finally` clause) would be cleaner.

Awakening a blocked task. There is no way for one task to awaken a second, blocked task, except by aborting it. Such a capability would sometimes be useful.

3.4 Implementability, Latency, and Efficiency

There are two basic approaches to implementing the asynchronous select statement, referred to as the "one thread" and "two thread" models [6, 7, 8]. In the one-thread model, the task does not block after executing the triggering statement but instead proceeds to execute the abortable part. The task is thus in a somewhat schizophrenic state of being queued on an entry or a timeout while still running. If the triggering statement occurs, the task is interrupted; it performs the necessary finalizations and then resumes in the triggering alternative. These effects can be implemented via the equivalent of a signal and `setjmp/longjmp`.

With the two-thread model, the task executing the asynchronous select is blocked at the triggering statement (as suggested by the syntax), and a subsidiary task is created to execute the abortable part. If the triggering condition occurs, the effect is equivalent to aborting the subsidiary task. Conceptually, this model seems cleaner than the one-thread approach. However, adding implicit threads of control is not necessarily desirable; e.g., data declared in the outer task and referenced in the abortable part would need to be specified as `Volatile` or `Atomic` to prevent unwanted caching. Moreover, aborting the subsidiary thread is complicated, due to the required semantics for execution of finalizations [7, 8].

As a result of these considerations, current Ada implementations generally use the one-thread model.

In addition to the basic question of how to implement the ATC semantics there are issues of run-time responsiveness (i.e., low latency) and efficiency. As noted above, the effect of ATC is sometimes deferred, in the interest of safety. If the abortable part declares any controlled objects or local tasks, or executes a rendezvous, then this will introduce a corresponding latency. The amount incurred is thus a function of programming style.

The efficiency question is somewhat different. Unlike its "pay as you go" effect on latency, ATC imposes a "distributed overhead"; i.e., there is a cost even if ATC isn't used. As an example, the epilog code for rendezvous and protected operations needs to check is there is a pending ATC request (either an abort or the completion of a triggering statement). Somewhat anticipating this issue, the Ada language allows the user to specify restrictions on feature usage, thus allowing/directing the implementation to use a more specialized and more efficient version of the run-time library, omitting support for unused features.

4 ATC in Java

This section briefly summarizes and critiques the ATC capabilities provided in Java [9] itself;[6] Java's asynchrony facilities inspired the RTSJ in either what to do or what not to do.

Java's support for asynchronous communication is embodied in three methods from the `Thread` class: `interrupt`, `stop`, and `destroy`. It also has a limited form of timeout via overloaded versions of `Object.wait`.

When `t.interrupt()` is invoked on a target thread[7] `t`, the effect depends on whether `t` is blocked (at a call for `wait`, `sleep`, or `join`). If so, an ATC takes place: an `InterruptedException` is thrown, awakening `t`. Otherwise, `t`'s "interrupted" state is set; it is reset either when `t` next calls the `interrupted` method or when it reaches a call on `wait`, `sleep`, or `join`. In the latter cases an `InterruptedException` is thrown.

[6] This description is based on [10].

[7] In this section "thread" means an instance of `java.lang.Thread`.

Despite the ATC aspects of `interrupt`, it is basically used in polling approaches: each time through a loop, a thread can invoke `interrupted` to see if `interrupt` has been called on it, and take appropriate action if so.

When `t.stop()` is invoked on a target thread `t`, a `ThreadDeath` exception is thrown in `t` wherever it was executing, and normal exception propagation semantics apply. This method was designed to terminate `t` while allowing it to do cleanup (via `finally` clauses as the exception propagates). However, there are several major problems:

- If `t.stop()` is called while `t` is executing synchronized code, the synchronized object will be left in an inconsistent state.
- A "catch-all" handler (e.g. for `Exception` or `Throwable`) in a `try` statement along the propagation path will catch the `ThreadDeath` exception, preventing `t` from being terminated.

As a result of such problems, the `Thread.stop()` method has been deprecated.

When `t.destroy()` is invoked, `t` is terminated immediately, with no cleanup. However, if `t` is executing synchronized code, the lock on the synchronized object will never be released. For this reason, even though `destroy` has not been officially deprecated, its susceptibility to deadlock makes it a dangerous feature. In any event, `destroy` has not been implemented in any JVM released by Sun.

Java's timeout support is limited to several overloaded versions of class `Object`'s `wait` method. However, there is no way to know (after awakening) which condition occurred: the timeout, or an object notification. Indeed, there are race conditions in which both may have happened; an object notification may occur after the timeout but before the thread is scheduled.

5 An Overview of the Real-Time Specification for Java

ATC is one of several facilities provided by the RTSJ. To establish a perspective, this section summarizes the problems that the RTSJ sought to address, and the main aspects of its solution.

The Real-Time Specification for Java is a class library (the package `javax.realtime`) that supplements the Java platform to satisfy real-time requirements. It was designed to particularly address the following shortcomings in regular Java:

Incompletely specified thread model. Java places only loose requirements on the scheduler.[8] There is no guarantee that priority is used to dictate which thread is chosen on release of a lock or on notification of an object. Priority inversions may occur; moreover, the priority range is too narrow.

Garbage Collector interference. Program predictability is compromised by the latency induced by the Garbage Collector.

Lack of low-level facilities. Java (although for good reasons) prevents the program from doing low-level operations such as accessing physical addresses on the machine.

Asynchrony shortcomings. As mentioned above, Java's features for asynchronous thread termination are flawed, and it lacks a general mechanism for timeouts and other asynchronous communication.

The RTSJ provides a flexible scheduling framework based on the `Schedulable` interface and the `Thread` subclass `RealtimeThread` that implements this interface. The latter class overrides various methods with versions that add real-time functionality, and supplies new methods for operations such as periodic scheduling. The `Schedulable` interface is introduced because certain schedulable entities (in particular, handlers for asynchronous events) might not be implemented as threads.

The RTSJ mandates a default POSIX-compliant preemptive priority-based scheduler that supports at least 28 priority levels, and that enforces Priority Inheritance as the way to manage priority inversions. The implementation can provide other schedulers (e.g., Earliest Deadline First) and priority inversion control policies (e.g., Priority Ceiling Emulation).

[8] This lack of precision may seem strange in light of Java's well-publicized claim to portability ("Write Once, Run Anywhere"). However, in the threads area there is considerable variation in the support provided by the operating systems underlying the JVM implementations. If the semantics for priorities, etc., were tighter, that would make Java difficult or inefficient to implement on certain platforms.

To deal with Garbage Collection issues, the RTSJ provides various *memory areas* that are not subject to Garbage Collection: "immortal memory", which persists for the duration of the application; and "scoped memory", which is a generalization of the run-time stack. Restrictions on assignment prevent dangling references. The RTSJ also provides a `NoHeapRealtimeThread` class; instances never reference the heap, may preempt the Garbage Collector at any time (even when the heap is in an inconsistent state), and thus do not incur GC latency.

The RTSJ provides several classes that allow low-level programming. "Peek and poke" facilities for integral and floating-point data are available for "raw memory", and "physical memory" may be defined with particular characteristics (such as flash memory) and used for general object allocation.

Java's asynchrony issues are addressed through two main features. First, the RTSJ allows the definition of asynchronous events and asynchronous event handlers – these are basically a high-level mechanism for handling hardware interrupts or software "signals". Secondly, the RTSJ extends the effect of `Thread.interrupt` to apply not only to blocked threads, but also to real-time threads[9] and asynchronous event handlers whether blocked or not. How this is achieved, and how it meets various software engineering criteria, will be the subject of the next section.

6 ATC in the Real-Time Specification for Java

This section describes, illustrates and critiques the RTSJ's ATC facilities.

6.1 Semantics

ATC in the RTSJ is defined by the effects of an asynchronous exception that is thrown in a thread[10] `t` as the result of invoking `t.interrupt`. However, asynchronous exceptions raise a number of issues that need to be resolved:

Inconsistent state. If the exception is thrown while the thread is synchronized on an object – or more generally, while the thread is in a code section that needs to be executed to completion – then the object (or some global state) will be left inconsistent when the exception is propagated.

Unintended non-termination. If the purpose of the ATC request is to terminate `t`, but the resulting exception is thrown while `t` is in a `try` block that has an associated catch clause for, say, `Exception` or `Throwable`, then the exception will be caught; `t` will not be terminated.

Unintended termination. If the purpose of the ATC request is, say, to make `t` timeout on a computation, but the exception is thrown either before `t` has entered a `try` statement with an associated handler, or after it has exited from such a construct, then the exception will not be handled. It will propagate out, and eventually cause `t` to terminate.

Nested ATCs / competing exceptions. A thread may receive an ATC request while another ATC is in progress. This raises the issue of propagating multiple exceptions or choosing which one should be discarded.

Indeed, these kinds of problems motivated the removal of the asynchronous `'Failure` exception from an early pre-standard version of Ada.

The RTSJ's approach to ATC addresses all of these issues. It is based on the class `AsynchronouslyInterruptedException`, abbreviated "`AIE`", a subclass of the checked exception class `InterruptedException`. An ATC request always involves, either explicitly or implicitly, a target thread `t` and an `AIE` instance `aie`. For example, the method call `t.interrupt()` posts an ATC request to the explicit target thread `t`, but the `AIE` instance (the system-wide "generic" `AIE`) is implicit.

Key to the semantics are the complementary concepts of *asynchronously interruptible* (or *AI*) and *ATC-deferred* sections. The only code that is asynchronously interruptible is that contained textually within a method or constructor that includes `AIE` on its `throws` clause, but that is not within synchronized

[9] A *real-time thread* is an instance of the `RealtimeThread` class.

[10] For ease of exposition, we refer to the target of `interrupt` as a "thread", but in fact for ATC effects it must be a real-time thread or asynchronous event handler. Regular Java threads – instances of `java.lang.Thread` – do not have ATC semantics.

code or in inner classes, methods, or constructors. Synchronized statements and methods, and also methods and constructors that lack a `throws AIE` clause, are ATC-deferred.

Posting an ATC request for `AIE` instance `aie` on thread `t` has the following effect:

1. `aie` is made *pending* on `t`.[11]
2. If `t` is executing within ATC-deferred code, `t` continues execution until it either invokes an AI method, or returns to an AI context.
3. If `t` is executing within AI code, `aie` is thrown (but stays pending).

Note that if control never reaches an AI method, then `aie` will stay pending "forever"; it will not be thrown.

If/when control does reach an AI method, then `aie` is thrown. However, the rules for handling `AIE` are different from other exceptions; this exception class is never handled by catch clauses in AI code. Instead, control transfers immediately – without executing `finally` clauses in AI code as the exception is propagated – to the catch clause for `AIE` (or any of its ancestor classes) of the nearest dynamically enclosing `try` statement that is an an ATC-deferred section. Unless the handling code resets the "pending" status, the `AIE` stays pending.

These rules address two of the previously-noted issues with asynchronous exceptions (and thus avoid the problems with `Thread.stop`):

- Since AI code needs to be explicitly marked with a `throws AIE` clause, and synchronized code is ATC-deferred, the RTSJ prevents inconsistent state (or at least forces the programmer to make explicit the possibility of inconsistent state).
- Since handling the `AIE` does not automatically reset the "pending" status, the RTSJ rules prevent unintended non-termination.

Since the RTSJ provides no mechanism for immediately terminating a thread, it avoids the difficulties inherent in `Thread.destroy`.

Here is an example of typical RTSJ style for thread termination:

```
class Victim extends RealtimeThread{
  private void interruptibleRun()
    throws AsynchronouslyInterruptedException{
    ... // Code that is asynchronously interruptible
  }

  public void run(){
    try{
      this.interruptibleRun();
    }
    catch (AsynchronouslyInterruptedException aie){
     System.out.println( "terminating" );
    }
  }
}
```

To create, start, and eventually terminate a `Victim` thread:

```
Victim v = new Victim();
v.start();
...
v.interrupt();
```

The immediacy of the effect of `v.interrupt()` depends on where `t` is executing when the ATC is posted. There are several possibilities:

[11] A special case, when there is already an `AIE` pending on `t`, is treated below.

- *Before reaching the invocation of* `interruptibleRun`. The generic `AIE` remains pending until `interruptibleRun` is invoked, and it is thrown then (without entering the called method) since `interruptibleRun` is AI. The catch clause in `run` handles the exception and displays the `"terminating"` message. The catch clause does not reset the `AIE`'s pending status, so the `AIE` stays pending[12] when `run` returns.
- *During execution of* `interruptibleRun`. The generic `AIE` is thrown either immediately (if control is not in an ATC-deferred section, such as a `synchronized` statement within the method's body) or else as soon as control is in AI code.
- *After returning from* `interruptibleRun`. The generic `AIE` stays pending on `t`, and the `run` method returns without the `AIE` having been thrown.

The declaration of an `AIE`-throwing method that is invoked from `run` is essential to allow the thread to be terminated from outside. If we simply declared a `run` method – note that Java semantics prohibit the inclusion of a `throws` clause for a checked exception class such as `AIE` – then invoking `t.interrupt()` would leave the generic `AIE` pending; the exception would never be thrown.

In order to address the problem of unintended termination – i.e., throwing an `AIE` outside the target thread's `try` statement that supplies a handler – the `AIE` class declares the `doInterruptible` and `fire` instance methods. The invocation `aie.doInterruptible(obj)` in a thread `t` takes an instance `obj` of a class that implements the `Interruptible` interface. This interface declares (and thus `obj` implements) the `run` and `interruptAction` methods. The `run` method (which should have a `throws AIE` clause) is invoked synchronously from `doInterruptible`. If `aie.fire()` is invoked – presumably from another thread that holds the `aie` reference – then an ATC request for `aie` is posted to `t`. The implementation logic in `doInterruptible` supplies a handler that invokes `obj.interruptAction` and resets `aie`'s *pending* state. If `aie.fire()` is invoked when control is not within `aie.doInterruptible` then the ATC request is discarded – thus `t` will only receive the exception when it is executing in a scope that can explicitly handle it (through user-supplied code).

A timeout is achieved by combining an asynchronous event handler, a timer, and an `AIE`. Since the details can be tedious, the RTSJ supplies the `Timed` subclass of `AIE`; a constructor takes a `HighResolutionTime` value. The application can then perform a timeout by invoking `timed.doInterruptible(obj)` where `timed` is an instance of `Timed`, and `obj` is an `Interruptible`. Here is an example, with the same effect as the Ada version in Section 3.1. It assumes that a `Position` object (passed in a constructor) is updated via the synchronized method `update`. Instead of declaring an explicit class that implements the `Interruptible` interface, it uses an anonymous inner class.

```
class Sensor extends RealtimeThread{
  final Position pos;
  final long sleepInterval  =  1000;
  final long timeout        = 10000;

  Reporter(Position pos){ this.pos = pos; }

  public void run(){
    new Timed(new RelativeTime( timeout, 0 )).
      doInterruptible(
        new Interruptible(){
          public void run(AsynchronouslyInterruptedException e)
             throws AsynchronouslyInterruptedException{
             while (true){
               pos.update();  // synchronized method
               try {
                 sleep(sleepInterval);
               }
               catch(InterruptedException ie) {}
```

[12]This doesn't matter here; in other contexts it may be important to make the `AIE` non-pending, and the `AIE` class supplies a method that has this effect.

```
            }
          }
          public void interruptAction(
            AsynchronouslyInterruptedException e){
              System.out.println("Sensor instance terminating");
          }
        });
  }
}
```

If the timeout occurs while the `Interruptible`'s `run()` method is in progress, execution of this method is abandoned (but deferred if execution is in synchronized code) and `interruptAction` is invoked, here simply displaying a termination message.

Nested ATCs raise the issue of multiple ATC requests in progress simultaneously. The RTSJ addresses this problem by permitting a thread to contain at most one pending `AIE`, and by defining an ordering relation between `AIE` instances. The rules give a "precedence" to an `AIE` based on the dynamic level of its "owning" scope, where ownership is established through an invocation of `doInterruptible`. Shallower scopes have precedence over deeper ones, and the generic `AIE` has the highest precedence. Thus a pending `AIE` is replaced by a new one only if the latter is "aimed" at a shallower scope.

6.2 Safety

Like Ada, the RTSJ opts for safety over immediacy and thus defers ATC in synchronized code (the Java analog to Ada's protected operations and rendezvous). However, ATC is not deferred in `finally` clauses,[13] thus leading to potential problems where essential finalization is either not performed at all or else only done partially.

Since the RTSJ controls asynchronous interruptibility on a method-by-method basis (i.e., via the presence or absence of a `throws AIE` clause), legacy code that was not written to be asynchronously interruptible will continue to execute safely even if called from an asynchronously-interruptible method.

6.3 Style and Expressiveness

A major issue with the RTSJ is the rather complex style that is needed to obtain ATC. Some examples:

- Aborting a thread requires splitting off an `AIE`-throwing method that is invoked from `run` (`interruptibleRun` in the `Victim` example above).
- Achieving timeout is somewhat obscure, typically involving advanced features such as anonymous inner classes or their equivalent.
- Programming errors that are easy to make – e.g., omitting the `throws AIE` clause from the `run` method of the anonymous `Interruptible` object – will thwart the ATC intent. This error would not be caught be the compiler.
- If the `run` method for an `Interruptible` invokes `sleep`, then the method has to handle `InterruptedException` even though the RTSJ semantics dictate that such a handler will never be executed.

These stylistic problems are due somewhat to the constraint on the RTSJ to not introduce new syntax. ATC is a control mechanism, and modeling control features with method calls tends to sacrifice readability and program clarity.

Another issue with the RTSJ is its mix of high-level and low-level features. An advantage is generality, but the disadvantage is that the programmer needs to adhere to a fairly constrained set of idioms in order to avoid writing hard-to-understand code.

The rules for `AIE` propagation diverge from regular Java semantics; this does not help program readability. For example, if an asynchronously interruptible method has a `try` statement with a `finally` clause, then the `finally` clause is not executed if an ATC is triggered while control is in the `try` block.

[13] This is because the RTSJ was designed to affect only the implementation of the Java Virtual Machine, and not the compiler. The syntactic distinction for the `finally` clause is not preserved in the class file, and there is no easy way for the implementation to recognize that such bytecodes should be ATC-deferred.

For situations where `t.interrupt()` is invoked to cause `t` to terminate, the method name "`interrupt`" is slightly misleading.

6.4 Implementability, Latency, and Efficiency

The conceptual basis for implementing ATC is a slot in each "thread control block" to hold an `AIE`, a flag indicating whether the `AIE` is pending, and a flag in each stackframe indicating if the method is currently asynchronously interruptible. Posting an `AIE` instance `aie` to a real-time thread `t` involves setting `aie` as `t`'s `AIE` (subject to the precedence rules) and setting the pending flag to `true`. Entering / leaving synchronized code affects the AI flag. The bytecodes for method call/return and exception propagation use these data values to implement ATC semantics.

As with Ada, the latency to asynchronous interruption depends on style.

Also as with Ada, ATC incurs a cost even if not used; the implementation of various bytecodes needs to check the ATC data structures.

7 Comparison

Table 1 summarizes how Ada and the RTSJ compare with respect to specific ATC criteria; for completeness it also includes the basic Java asynchrony mechanisms. The following subsections will elaborate on the Ada and RTSJ entries.

Table 1: *Comparison of ATC Mechanisms*

	Ada		**Java**			**RTSJ**
	abort	*asynch select*	*interrupt*	*stop*	*destroy*	*AIE*
Semantic basis	Task abort		Synchronous exception	Asynch exception	Immediate terminate	Asynch exception
Safety	Good	Good	Good	Poor	Poor	Good
Defer in synchronized code	Yes	Yes	No	No	No	Yes
Defer in finalization	Yes	Yes	No	No	No	No
Defer unless explicit	No	No	No	No	No	Yes
Style	Good	Good	Good	Poor	Poor	Fair
Expressiveness	Good	Good	Fair	Fair	Poor	Good
Implementability	Good	Fair	Good	Fair	Poor	Fair
Latency	Fair	Fair	Poor	Good	Good	Fair
Efficiency	Fair	Fair	Good	Fair	Poor	Fair

7.1 Semantics

Ada and the RTSJ take very different approaches to ATC. Ada defines the semantics for asynchronous task termination (the `abort` statement) and then applies these semantics in another context (the asynchronous select) to model ATC triggered by a timeout or the servicing of an entry call. Ada does not define ATC in terms of exceptions. Indeed, aborting a task `t` does not cause an exception to be thrown in `t`; even if a `Finalize` procedure throws an exception, this exception is not propagated ([1], ¶7.6.1(20)).

In contrast, the RTSJ's ATC mechanism is based on asynchronous exceptions, a somewhat natural design decision given the semantics of Java's `interrupt()` facility. Thus the RTSJ has a general ATC approach, and realizes real-time thread termination as a special case of ATC. However, a side effect is a rather complicated set of rules, e.g., the precedence of exceptions.

7.2 Safety

Both Ada and the RTSJ recognize the need to define regions of code where ATC is inhibited, in particular in code that is executed under a "lock". Ada is safer in specifying additional operations (e.g. finalization of controlled objects) as abort-deferred; in the RTSJ `finally` clauses are asynchronously interruptible.

The RTSJ, however, offers finer granularity of control than Ada; asynchronous interruptibility can be specified on a method-by-method basis, and the default is "noninterruptible". It thus avoids the problem of aborting code that might not have been written to be abortable.

7.3 Style and Expressiveness

Ada and the RTSJ are roughly comparable in their expressive power but they differ significantly with respect to their programming styles. Ada provides distinguished syntax – the `abort` and asynchronous select statements – whose effects are clear. The RTSJ realizes ATC via method calls, which sacrifices readability, as is evidenced by comparing the two versions of the `Sensor` timeout example. There are also a number of non-intuitive points of style that programmers will need to remember.

On the other hand, Java's `interrupt()` mechanism allows the programmer to awaken a blocked thread by throwing an exception; Ada lacks a comparable feature. Moreover, the RTSJ is somewhat more regular than Ada with respect to feature composition. ATC-deferred code can call an AI method, and an ATC can thus be triggered in the called method. In Ada it is a bounded error if a subprogram invoked by an abort-deferred operation attempts an asynchronous select statement.

7.4 Implementability, Latency and Efficiency

ATC in both Ada and the RTSJ requires non-trivial implementation support; these features are complicated semantically (in the details if not the main concepts) and are among the most difficult to implement and to test.

ATC latency is roughly equivalent in Ada and the RTSJ, and is a function of programming style. Heavy use of abort/ATC-deferred constructs will induce high latency, and inversely. However, since subprograms by default are abortable in Ada, the latency from ATC in Ada is likely to be less than in the RTSJ.

Efficiency will be a challenge for both Ada and the RTSJ; this seems intrinsic in ATC rather than a flaw in either design. Non-optimizing implementations will likely impose an overhead even if ATC is not used. Possible approaches include sophisticated control and data flow analysis, hardware support, and the definition of restricted profiles.

8 Conclusions

ATC is a difficult issue in language design, and both Ada and the RTSJ have made serious attempts to provide workable solutions. They share a common philosophy in opting for safety as the most important objective, and thus in defining ATC to be deferred in certain regions that must be executed to completion. They offer roughly comparable expressive power, but they differ significantly in how the mechanism is realized and in the resulting programming style. Ada bases ATC on the concept of abortable code regions, integrates ATC with the inter-task communication facility, and provides specific syntax for ATC in general and for task termination in particular. The RTSJ bases ATC on an asynchronous exception thrown as the result of invoking `interrupt()` on a real-time thread; termination is a special case. The RTSJ does not introduce new syntax for ATC, so the effect must be achieved through new classes and method calls.

Since the RTSJ is so new, there is not much experience revealing how it compares with Ada in practice. As implementations mature, and developers gain familiarity with the concepts, it will be interesting to see whether ATC fulfills its designers' expectations and its users' requirements.

A An ATC Example

This Appendix presents a complete example that illustrates thread termination and timeout, and provides sample solutions in Ada and Java/RTSJ.

A.1 Problem Statement / Program Requirements

An application comprises a shared data object and three threads of control:

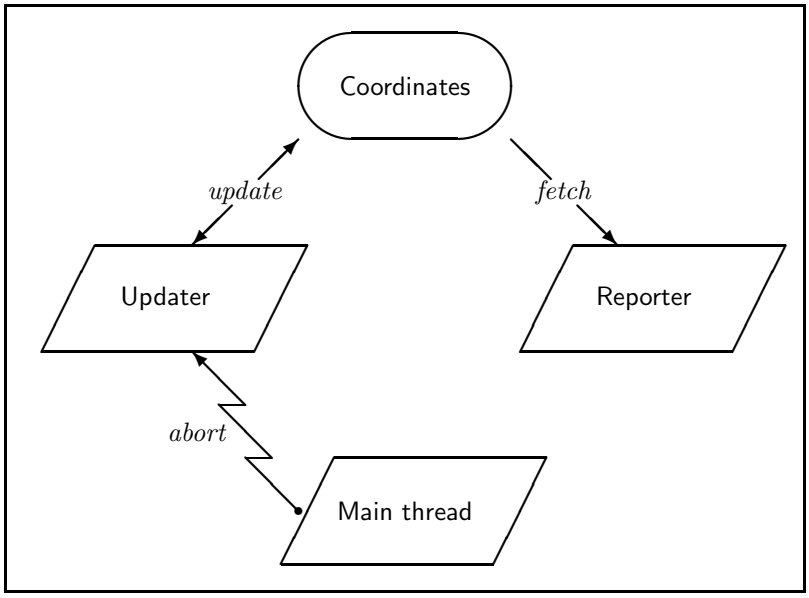

Figure 1: *ATC Example*

A "coordinates" object. This object is an array of two integers. Two operations are permitted: *update*, which adds 1 to each element of the array; and *fetch*, which returns the current value for the array. Each operation requires mutually exclusive access to the object.[14]

An "updater" thread. This thread invokes the *update* operation on the coordinates object at intervals no shorter than 2 seconds.[15] It runs "forever", or until its execution is aborted by some other thread of control. In the latter case, the updater thread should display the message `"Updater terminating"` before it terminates. Aborting the updater thread must not leave the coordinates object inconsistent: i.e., if the thread is aborted while executing the *update* operation, then the *update* must be allowed to complete.

A "reporter" thread. This thread *fetch*es and then displays the value of the coordinates object, at intervals no shorter than 1 second. It times out after 10 seconds, after which it displays the message `"Reporter terminating"` and then terminates.

The main thread of control. This thread creates the coordinates object and activates the other two threads, after which it suspends for 5 seconds and then aborts the execution of the updater thread.

Figure 1 indicates the relationships among these components. The straight arrows show the data flows between the data object and the threads; *update* both "reads" and "writes" the data object, and *fetch* only "reads" it. The jagged arrow depicts asynchronous communication between threads, here an *abort* ATC request. The timeout ATC internal to the reporter thread is not shown.

A.2 Ada Version

```
1  with Ada.Finalization;
2  package Last_Wishes_Pkg is
3    type Last_Wishes is new Ada.Finalization.Controlled
4      with null record;
5    procedure Finalize( Item : in out Last_Wishes );
6  end Last_Wishes_Pkg;
7
```

[14] Actually the *fetch* operation only needs a "read" lock, but this detail is not critical to the example.

[15] For simplicity, we are not concerned with more precise periodicity. If we were, then both Ada and the RTSJ have relevant features for addressing this issue.

```
 8  with Ada.Text_IO; use Ada.Text_IO;
 9  package body Last_Wishes_Pkg is
10    procedure Finalize( Item : in out Last_Wishes ) is
11    begin
12      Put_Line("Updater terminating");
13    end Finalize;
14  end Last_Wishes_Pkg;
15
16  with Last_Wishes_Pkg, Ada.Text_IO;
17  use Last_Wishes_Pkg, Ada.Text_IO;
18  procedure Ada_ATC is
19
20    type Pair is array(1..2) of Integer;
21
22    protected Coordinates is
23      procedure Update;
24      function Fetch return Pair;
25    private
26      Data : Pair := (0, 0);
27    end Coordinates;
28
29    protected body Coordinates is
30      procedure Update is
31      begin
32        for I in Data'Range loop
33          Data(I) := Data(I) +1;
34        end loop;
35      end Update;
36
37      function Fetch return Pair is
38      begin
39        return Data;
40      end Fetch;
41    end Coordinates;
42
43    task Updater;
44
45    task body Updater is
46      Period : constant Duration := 2.0;
47      LW     : Last_Wishes;
48    begin
49      loop
50        Coordinates.Update;
51        delay Period;
52      end loop;
53    end Updater;
54
55    task Reporter;
56
57    task body Reporter is
58      Time_Out : constant Duration := 10.0;
59      Period   : constant Duration := 1.0;
60      Data     : Pair;
61    begin
62      select
63        delay Time_Out;
```

```
64        Put_Line("Reporter terminating");
65      then abort
66        loop
67          Data := Coordinates.Fetch;
68          Put( "(" & Integer'Image(Data(1)) & "," &
69                     Integer'Image(Data(2)) & ") " );
70          delay Period;
71        end loop;
72      end select;
73    end Reporter;
74  begin
75    delay 5.0;
76    abort Updater;
77  end Ada_ATC;
```

The updater and reporter threads are directly modeled by tasks local to the main procedure, and the coordinates variable is captured by a protected object.

The `Updater` task declares a local controlled variable `LW` (line 47), to arrange appropriate finalization (here simply displaying a message) if the task is aborted.

The `Reporter` task implements the timeout via an asynchronous select statement (lines 62–72). There is an implicit assumption that the `Put` procedure is safe to abort (or that the run-time library will defer the timeout until a safe abort point is reached). If this assumption is not correct, the invocation of `Put` needs to be placed in an abort-deferred construct. A simple way to do this (at least in the GNAT implementation) is to enclose the call of `Put` within a block that has an `Abort_Defer` pragma:

```
begin
  pragma Abort_Defer;
  Put( "(" & Integer'Image(Data(1)) & "," &
             Integer'Image(Data(2)) & ") " );
end;
```

The program produces the following sample output (GNAT 3.15a on Windows):

```
( 1, 1) ( 1, 1) ( 2, 2) ( 2, 2) ( 3, 3) Updater terminating
( 3, 3) ( 3, 3) ( 3, 3) ( 3, 3) ( 3, 3) Reporter terminating
```

A.3 RTSJ version

```
 1  class Coordinates{
 2    final private int[] data = {0, 0};
 3    synchronized public void update(){
 4       data[0]++;
 5       data[1]++;
 6    }
 7    synchronized public void fetch(int[] data){
 8      data[0] = this.data[0];
 9      data[1] = this.data[1];
10    }
11  }
12
13  class RtsjAtc{
14    public static void main( String[] args ) throws InterruptedException{
15      final Coordinates coordinates = new Coordinates();
16
17      final Reporter reporter = new Reporter(coordinates);
18      final Updater  updater  = new Updater(coordinates);
19
```

```
20      reporter.start();
21      updater.start();
22      Thread.sleep(5000);
23      updater.interrupt();
24    }
25  }
26
27  import javax.realtime.*;
28  class Updater extends RealtimeThread{
29    final Coordinates coordinates;
30    final long period = 2000;
31
32    Updater(Coordinates coordinates){
33      this.coordinates = coordinates;
34    }
35
36    private void interruptibleRun()
37      throws AsynchronouslyInterruptedException{
38      while (true){
39        coordinates.update();
40        try {
41          sleep( period );
42        }
43        catch(InterruptedException ie) {}
44      }
45    }
46
47    public void run(){
48      try{
49        this.interruptibleRun();
50      }
51      catch (AsynchronouslyInterruptedException aie){
52       System.out.println( "Updater terminating" );
53      }
54    }
55  }
56
57  import javax.realtime.*;
58  class Reporter extends RealtimeThread{
59    final Coordinates coordinates;
60    final long period  =  1000;
61    final long timeout = 10000;
62
63    Reporter(Coordinates coordinates){
64      this.coordinates = coordinates;
65    }
66
67    public void run()
68    {
69      new Timed(new RelativeTime( timeout, 0 )).
70        doInterruptible(
71          new Interruptible(){
72            public void run(AsynchronouslyInterruptedException e)
73               throws AsynchronouslyInterruptedException{
74               int[] data = new int[2];
75               while (true){
```

```
76                 coordinates.fetch(data);
77                 System.out.print("(" + data[0] + ","
78                                      + data[1] + ") ");
79                 try {
80                   sleep(period);
81                 }
82                 catch(InterruptedException ie) {}
83               }
84             }
85             public void interruptAction(
86               AsynchronouslyInterruptedException e){
87                 System.out.println("Reporter terminating");
88             }
89           }
90           );
91     }
92  }
```

The `main()` method in the `RtsjAtc` class is executed by the main program thread. This method constructs the `coordinates` object that is shared by the reporter and updater threads, and then constructs and starts these threads.

The logic of the updater thread is separated into two methods: `interruptibleRun` and `run`.

- The `interruptibleRun` method is asynchronously interruptible, since it specifies `AsynchronouslyInterruptedException` on its `throws` clause. When its execution is interrupted by the main thread's call `updater.interrupt()`, an instance of `AsynchronouslyInterruptedException` will be propagated back to the caller. However, since ATC is deferred in synchronized code, this exception will not be thrown when `updater` is executing `coordinates.update()`.
- The `run()` method is invoked when `updater` is started. Its catch clause handles the asynchronous exception propagated by `interruptibleRun`.

The `reporter` thread illustrates how the RTSJ models a timeout. The `doInterruptible` method is invoked on an instance of the `Timed` class. This method sets up the timeout framework by constructing an `Interruptible` object that provides an asynchronously interruptible `run()` method and an `interruptAction` "handler". If the timeout occurs while this `run()` method is in progress, execution of the `run()` method is abandoned. Note, however, that since `fetch` is a synchronized method, and `print` does not include `AIE` on its `throws` clause, neither of these methods is asynchronously interruptible. Thus `run` will only be interrupted when control is not within either of these methods. After execution of `run` is abandoned, `interruptAction` is invoked; it will display a termination message.

The program produces the following sample output (TimeSys Reference Implementation):

```
(1,1) (1,1) (2,2) (2,2) (3,3) Updater terminating
(3,3) (3,3) (3,3) (3,3) (3,3) Reporter terminating
```

A.4 Discussion

Safety. Both solutions guarantee that protected (Ada) / synchronized (Java) code will not be aborted. There is the issue in the Ada solution that the invocation of `Put` is not abort-deferred; this requires the Ada developer to either check that such a call is safe, or else to place the call in a construct that is guaranteed to be abort deferred.

Style. On the whole, the Ada version is much clearer than the RTSJ version. Ada's specialized syntax for ATC makes the intent apparent: aborting the `Updater` task is direct (there is no need, as in the RTSJ version, to partition the processing into interruptible and noninterruptible parts). The difference between the timeout styles is even more pronounced. The asynchronous select statement clearly indicates

the intent and the control flow. In contrast, the RTSJ version requires constructing an instance of a class that implements `run` and `interruptAction` methods, and the effect is not nearly as apparent.

The one area where the RTSJ version is superior stylistically is in capturing the finalization processing. Ada requires the declaration of a library-level controlled type (`Last_Wishes`), and a never-referenced variable (`LW`) in the body of the `Updater` task, in order to arrange the necessary finalization for that task. The RTSJ version accomplishes the same effect much more simply in a `catch` clause within `Updater`'s `run` method.

Efficiency. Attempting to compare performance across languages and platforms (native versus JVM) is very difficult. We will simply point out that Ada has the advantage over Java/RTSJ in some areas (e.g., the compiler can more easily optimize specialized syntax than method calls) but incurs higher overhead in others (e.g., finalization and handling of task hierarchies).

References

[1] S.T. Taft, R.A. Duff, R.L. Brukardt, and E. Ploedereder; *Consolidated Ada Reference Manual, Language and Standard Libraries, International Standard ISO/IEC 8652/1995(E) with Technical Corrigendum 1*; Springer LNCS 2219; 2000.

[2] G. Bollella, J. Gosling, B. Brosgol, P. Dibble, S. Furr, D. Hardin, and M. Turnbull; *The Real-Time Specification for Java*, Addison-Wesley, 2000.

[3] The Real-Time for Java Expert Group; *The Real-Time Specification for Java, V1.0*; Sun Microsystems JSR-001; `http://www.rtj.org`; November 2001.

[4] B. Brosgol and A. Wellings; "A Comparison of the Asynchronous Transfer of Control Features in Ada and the Real-Time Specification for Java", *Proc. Ada Europe 2003*, June 2003, Toulouse, France.

[5] ISO/IEC 9945-1: 1996 (ANSI/IEEE Standard 1003.1, 1996 Edition); *POSIX Part 1: System Application Program Interface (API) [C Language].*

[6] E.W. Giering and T.P. Baker; "The GNU Runtime Library (GNARL): Design and Implementation", *WAdaS '94 Proceedings*, ACM SIGAda; 1994.

[7] E.W. Giering and T.P. Baker; "Ada 9X Asynchronous Transfer of Control: Applications and Implementation", *Proceedings of the SIGPLAN Workshop on Language, Compiler, and Tool Support for Real-Time Systems*, ACM SIGPLAN; 1994.

[8] J. Miranda; *A Detailed Description of the GNU Ada Run Time (Version 1.0)*; `http://www.iuma.ulpgc.es/users/jmiranda/gnat-rts/main.htm`; 2002.

[9] J. Gosling, B. Joy, G. Steele, G. Bracha; *The Java Language Specification (2nd ed.)*; Addison Wesley, 2000.

[10] B. Brosgol, R.J. Hassan II, and S. Robbins; "Asynchronous Transfer of Control in the Real-Time Specification for Java", *Proc. ISORC 2002 – The 5th IEEE International Symposium on Object-oriented Real-time distributed Computing*, April 29 - May 1, 2002, Washington, DC - USA.