Asynchronous Transfer of Control in the Real-Time Specification for Java™

Benjamin M. Brosgol
Ada Core Technologies, Inc.

Ricardo J. Hassan II
NASA Jet Propulsion Laboratories

Scott Robbins
TimeSys, Inc.

Abstract

The Real-Time Specification for Java provides a facility for Asynchronous Transfer of Control based on exception handling and a generalization of the `interrupt()` method from the `Thread` class. This mechanism allows the programming of useful idioms such as timeouts and thread termination without the latency found in polling, and it avoids the problems inherent in the `Thread` class's `stop()` and `destroy()` methods.

1 Introduction

Asynchronous Transfer of Control ("ATC") is a transfer of control within a thread, triggered not by the thread itself but rather from some action by another thread or an event handler. This facility was the most controversial element of the Real-Time Specification for Java ("RTSJ") [Bollella *et. al.* 2000] [JSR-1 2001]. Indeed, there are strong arguments against providing such a mechanism. ATC is methodologically suspect, since it is difficult to write correct code if control transfers can occur at unpredictable points. It is complicated to specify and to implement, and in the absence of compiler or JVM optimizations it may incur a performance penalty even for programs that don't use it. Nevertheless, ATC has been included for two main reasons. First, it allows expressing a number of common situations in real-time programs:

- Timing out on a computation (e.g. an iterative solution to a mathematical problem, where the intent is to obtain the most accurate result within some fixed amount of time)
- Terminating a thread
- Aborting one iteration of a loop

Second, the alternative to ATC – polling – may have an unacceptably high latency.

This paper describes the approach to ATC taken by the RTSJ. We first present the mechanisms available in Java[1] and explain their problems. We then summarize the

[1] In this paper "Java" means the Java language as defined in the *Java Language Specification* [Gosling *et. al.* 2000] and the `java.lang` package in the basic API [Chan *et. al.* 1998.] (i.e., without the real-time extensions)

principles and requirements on which the RTSJ's approach was based. We next explain the major elements of the RTSJ's approach, with supporting rationale for the key decisions and examples of typical idioms. We conclude by outlining a possible implementation model for ATC in a JVM.

2 ATC in Java

Java's `Thread` class provides several methods related to ATC: `interrupt()`, `stop()`, and `destroy()`.

When `t.interrupt()` is invoked on a target thread `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.

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 (such as returning from the `run()` method).

When `t.stop()` is invoked on a target thread `t`, a `ThreadDeath` exception is thrown at `t` wherever it was executing, and normal exception propagation semantics apply. This method was originally provided 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" hander (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 these problems, `stop()` 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 deprecated, its susceptibility to deadlock makes it a dangerous feature.

Several other Java facilities are related to ATC (more specifically, to thread termination). When `System.exit()` is invoked the JVM is halted, thus terminating any threads currently running. Somewhat similarly, after the last non-daemon thread terminates, any daemon threads are halted. Both of these situations can result in threads being terminated while engaged in operations that logically should not be aborted, and thus are potentially dangerous.

The RTSJ's approach to ATC was designed to avoid these problems. The next section summarizes the principles on which the facility is based.

3 Requirements and Principles for ATC[2]

3.1 Basic RTSJ Requirements

- Do not extend the Java syntax. Any effects must be expressible in terms of the grammar defined by the Java Language Specification [Gosling *et. al.* 2000]
- More generally, do not require compilers to be modified in order to implement the RTSJ. Although the RTSJ obviously affects the implementation of the JVM, developers should be able to use existing compilers for programs that import the `javax.realtime` package. (Nonetheless, certain RTSJ features may incur overhead unless implemented by compilers that perform specialized optimizations.)

3.2 Methodological Principles

- Susceptibility to ATC must be explicit in the source code (i.e., ATC is deferred in the absence of such an explicit indication).
- Even if it is explicitly enabled, ATC must be deferred in some code sections (in particular all synchronized code) in order to prevent shared objects from being left in an inconsistent state.
- Code that responds to an ATC does not return to the point where the ATC was triggered. (Otherwise it becomes very difficult to write a correct and time-predictable program with ATC, since in an asynchronously interruptible section the ATC response code could be invoked between any two bytecode instructions.)

3.3 Expressibility Principles

- A mechanism is needed through which an ATC can be explicitly triggered in a target thread. This triggering may be direct (from a source thread) or indirect (through an asynchronous event handler, e.g. for a timer).
- Through ATC it must be possible to abort a thread but in a manner that does not carry the dangers of the `Thread` class's `stop()` and `destroy()` methods.

3.4 Semantic Principles

- If ATC is modeled by exception handling, there must be some way to ensure that an asynchronous exception is only caught by the intended handler, and not, for example, by an all-purpose handler that happens to be on the propagation path.
- Nested ATC's (for example for timeouts) must work properly. A timeout from an outer timer must be handled in the outer scope, even if control is in the scope of an inner (longer) timer.

---

[2] This material is based on [Bollella *et. al.* 2000], pp. 2, 12-13

3.5 Pragmatic Principles

- There should be straightforward idioms for common cases such as timeouts and thread termination.
- ATC-related overhead should be minimal for programs that do not use ATC.
- If code with a timeout completes before the timeout's deadline, the timeout needs to be automatically stopped and its resources returned to the system.

4 ATC Design Decisions

4.1 Asynchronous Interruptibility

A fundamental decision was to require that a program explicitly indicate its susceptibility to ATC; in the parlance of the RTSJ, it needs to explicitly indicate where it is *asynchronously interruptible* ("AI"). An immediate issue is the granularity at which asynchronous interruptibility is specified. Thread level is too coarse, since a thread might need to be AI in some sections and not AI in others (such as while holding a lock). At the other extreme, having explicit methods such as enableAsynchronousInterruption and disableAsynchronousInterruption would be complicated to define (e.g., issues with nesting) and prone to error (calling enableAsynchronousInterruption and then not calling disableAsynchronousInterruption, perhaps because of an exception that was thrown before the matching disable operation was reached).

The compromise position taken by the RTSJ was to define asynchronous interruptibility at the level of the individual method or constructor. The way in which a method or constructor indicates that it is AI is via a throws clause that includes the exception class `AsynchronouslyInterruptedException` (abbreviated "AIE" hereafter)[3]. AIE is defined as a subclass of `InterruptedException`. A method or constructor lacking a "throws AIE" clause is not asynchronously interruptible. If an ATC is triggered while a realtime thread is in such code, the ATC is deferred until the thread next invokes or returns to an AI method or constructor.

ATC is deferred not only in methods or constructors lacking a "throws AIE" clause, but also in all synchronized code. This is essential to avoid the `Thread.stop()` and `Thread.destroy()` problems. A disadvantage of deferring ATC in synchronized code is that ATC cannot be used to break a (lock-based) deadlock[4]; however, the RTSJ considered it more important to keep synchronized code safe. Note that synchronized code may invoke an AI method; an ATC from the AI method gets transformed into a (synchronous) AIE exception propagated back to the invoking synchronized context.

Asynchronous interruptibility is defined only for realtime threads and not for threads in general. Thus a (non-realtime) thread invoking an AI method is not susceptible to ATC.

---

[3] The exception class identified in the throws clause needs to be exactly the class `AsynchronouslyInterruptedException` and not a subclass. This makes it clearer to the reader which methods are AI.

[4] When caused by interdependent invocations of `wait()` rather than synchronized code, a deadlock can be broken by an ATC.

During the evolution of the RTSJ other ATC-deferred code sections were proposed, for methodological reasons; in particular, constructors and `finally` clauses. The rationale for considering such sections to be ATC-deferred was that an ATC out of a constructor could cause inconsistent global state, and an ATC out of a `finally` clause could skip needed finalizations. However, since the programmer can decide (via the presence of a "throws AIE" clause) whether a constructor is AI, there is no need for the specification to be more constraining. And since `finally` clauses are not recognizable in the generated bytecodes, requiring such sections to be ATC-deferred would have affected the compiler, in violation of one of the RTSJ's guiding principles. Thus `finally` clauses and constructors are not defined as ATC-deferred sections (but of course a `finally` clause in a context that is not AI, and a constructor lacking a "throws AIE clause", are ATC deferred).

4.2 Underlying Model for ATC

There are two principal ways to specify ATC. One approach, illustrated by Ada 95 [Intermetrics 95], is based on the concept of aborting a thread. In this model there is a statement to abort a thread and there is also the notion of an *abort-deferred region* (e.g., code that is being executed by a thread that is holding a lock). ATC is captured in a special control structure. A section of code that is susceptible to ATC is conceptually a separate thread, associated with a triggering event such as a timeout. If the event occurs before the separate thread terminates, then the thread is aborted as soon as it is outside of an abort-deferred region. If the separate thread terminates before the event occurs, then the event is canceled. (This is a conceptual model only; it is possible to implement ATC within a single thread of control via a `setjmp` / `longjmp` approach. Also there are other details, such as the execution of finalization code before an aborted thread terminates, that we are omitting.)

There are several advantages to the two-thread model. Since it is based on the thread abort concept, no new mechanisms are needed for thread termination. It avoids the complications of an exception-based approach such as special rules for propagation (see below). And it deals with nested ATCs smoothly.

However, the RTSJ did not adopt this approach, for two main reasons. First, the overhead of thread management for ATC, and the complexity due to interactions with the RTSJ scheduling framework, would have been excessive in many situations. Second, the handling of nested ATC would have required introducing a concept of thread dependence (so that aborting a thread implicitly aborts all of its dependents) that is otherwise absent from Java and the RTSJ.

Instead, the RTSJ's ATC model is based on exceptions. Intuitively this seems a natural approach, especially since there is already an established syntax in Java for dealing with synchronous exceptions. However, capturing ATC by simply allowing a thread to arbitrarily throw an exception at a target thread (even with the requirement to defer throwing the exception when the target thread is in ATC-deferred code) would not work:

- If the exception is thrown before the target thread is ready to handle it (i.e., before the target thread reaches a `try` statement with an applicable catch clause) or after the target thread is no longer ready to handle it (i.e. after it has completed such a

`try` statement) then throwing the exception will have the effect of terminating the target thread since the exception will simply propagate out.

- If the exception is thrown while the target thread is executing a try statement with an all-purpose catch clause such as for `Exception` or `Throwable`, then it may incorrectly handle the asynchronous exception. In particular if the exception is supposed to lead to the termination of the target thread, then the accidental handling of this exception will prevent the target thread from terminating.

The key point is to ensure that an asynchronous exception is only thrown at a thread when the thread has dynamically indicated its readiness to deal with that exception.

The RTSJ supplies mechanisms at several levels to meet the requirements for ATC:

- Low-level building blocks for detailed control of exception propagation
- A higher-level construct for an idiom that hides the low-level details
- A class that specifically captures the use of ATC for timeout

The next section describes these three mechanisms as well as other aspects of ATC semantics.

5 ATC Semantics

This section presents the rules associated with ATC. These comprise both an API (classes and interfaces in the `javax.realtime` package) and an extension of the semantics of exception propagation. Several kinds of examples are presented, some to illustrate the semantics and others to show typical idioms.

5.1 ATC Building Blocks

ATC semantics are ultimately based on two concepts:

- The `interrupt()` method in class `RealtimeThread`
- The way in which an instance of AIE is propagated

The main ideas are relatively straightforward, although there are some differences (noted below) from regular Java exception semantics, and perhaps the idea of an exception staying pending even after it has been handled is somewhat novel. The principal complication arises from the possibility of one ATC being triggered while another one is in progress. Here we focus only on the case with one ATC active at a time; nested ATC is described in section 5.4.

5.1.1 Basic Properties

The AIE class has a specific instance, referred to as the "generic AIE instance" (or simply the "generic AIE"), which is returned by the static method `getGeneric()`. When `t.interrupt()` is invoked on a `RealtimeThread` `t`, this "generic" AIE becomes *pending* on `t`. There are two possibilities:

1. If `t` is in ATC-deferred code then it continues regular execution (which may entail throwing and handling other exceptions), until it enters AI code – either by

returning (normally or abnormally) to or calling an AI method. At this point the AIE is thrown but stays pending. This rule has several consequences. First, if `t` never invokes or returns to an AI method, then `t` will eventually terminate without the AIE ever being thrown. Second, if a non-AI method attempts to propagate a regular exception back to an AI method while an AIE is pending, then the regular exception is discarded and the AIE is thrown instead.

2. If `t` is in AI code then the AIE is thrown immediately. However, unlike other exceptions, an AIE is not caught by a `catch` clause in AI code. Instead, control transfers to the `catch` clause (for AIE or any of its ancestors) of the nearest dynamically enclosing `try` statement that is in an ATC-deferred section[5], and the AIE stays pending. It is important to realize that, as the stackframes for AI methods are peeled back, neither exception handlers nor `finally` clauses in these methods are executed. Note that there must be some such dynamically enclosing `try` statement, since AIE is a checked exception class and the `run()` method in `RealtimeThread` does not have a `throws` clause.

These rules make AIE semantics somewhat different from regular exceptions.

As with the `interrupt()` method in class `Thread`, invoking `t.interrupt()` on a `RealtimeThread` `t` will awaken `t` if `t` is blocked at a call of `wait()`, `join()`, or `sleep()`. The generic AIE will be thrown at that point and will be pending. Likewise, if `t` is executing in an ATC-deferred section when `t.interrupt()` occurs, then the generic AIE is thrown and stays pending when `t` calls one of these blocking methods. The effect is then as described above. Notice that the AIE will be thrown even if `t` is in ATC-deferred code. (Indeed, in order for `t` to invoke `wait()` it will need to be in ATC-deferred code since it requires the lock on the object that is `wait`'ed.)

The reason that `interrupt()` is defined to awaken a blocked thread is to allow the programmer to avoid unbounded blocking. A potential issue arises in connection with I/O calls. Attempting to identify in the RTSJ which I/O methods are unblocked when interrupted is not practical and indeed whether or not a call blocks (and, if it blocks, whether that is reasonably detectable) may be implementation dependent. The RTSJ's position is to require that methods in the `java.io` package be prevented from blocking indefinitely when invoked from an AI method. If `t.interrupt()` is invoked on a realtime thread `t` while `t` is in a method from a `java.io` class, then the effect is implementation dependent: `t` may throw an `IOException` or the generic `AsynchronouslyInterruptedException`, or it may alternatively be allowed to continue execution if the implementation can determine that unbounded blocking would not occur.

Handling an AIE does not make it non-pending. Once an AIE is triggered it will continue to propagate up the call chain, eventually terminating the thread, unless the program explicitly resets the AIE's pending status. This reset is generally performed by a non-AI method in a `catch` clause for AIE, through the method call

[5] Note that if the call of the interrupted AI method was from a synchronized block, then the relevant catch clause might not be in synchronized code, and thus the lock will be released prematurely. However, this is no different from the situation with synchronous exceptions propagated out of synchronized blocks.

`aie.happened(false)` where `aie` is the `catch` clause's exception parameter; the `false` argument indicates that propagation should not take place.

A note on `finally` clauses: If an AI method has a `try` statement with a `finally` clause, then the finally clause is executed only if no ATC is triggered while control is in the try statement. If application logic demands that the finally clause be executed regardless of whether an ATC is triggered, then the try statement should be taken out of line and placed in a non-AI method that is called from the AI method.

5.1.2 Example: AIE Semantics

Here is an artificial example that illustrates the semantics:

```
class Frammis extends RealtimeThread{
  public void run(){
    System.out.println("Entering try in run");
    notAI1();
    System.out.println("Finishing try in run");
  }

  void notAI1(){
    try{
      System.out.println("Entering try in notAI1");
      yesAI1();
      System.out.println("Finishing try in notAI1");
    }
    catch (AsynchronouslyInterruptedException aie){
      System.out.println("Catching AIE in notAI1");
      aie.happened(false);  // Reset pending status
    }
  }

  void yesAI1() throws AsynchronouslyInterruptedException{
    try{
      System.out.println("Entering try in yesAI1");
      yesAI2();
      System.out.println("Finishing try in yesAI1");
    }
    catch (Exception e){
      System.out.println("Catching Exception in yesAI1");
    }
    finally {
      System.out.println("In yesAI1, finally");
    }
  }

  void yesAI2() throws AsynchronouslyInterruptedException{
    try{
      System.out.println("Entering try in yesAI2");
      notAI2();
      System.out.println("Finishing try in yesAI2");
    }
    catch (Exception e){
      System.out.println("Catching Exception in yesAI2");
    }
    finally {
```

```
      System.out.println("In yesAI2, finally");
    }
  }
  void notAI2(){
    try{
      System.out.println("Entering try in notAI2");
      throw new RuntimeException();
    }
    finally{
      System.out.println("In notAI2, finally");
    }
  }
  public static void main(String[] args){
    Frammis f = new Frammis();
    f.start();
    ...
    f.interrupt();
    ...
  }
}
```

In this example the realtime thread `f` invokes a non-AI method (`run`), which invokes another non-AI method (`notAI1`), which invokes an AI method (`yesAI1`), which invokes another AI method (`yesAI2`), which invokes another non-AI method (`notAI2`). The effect depends on where `f` is executing when `f.interrupt()` is invoked in `main`.

- `Run / notAI1 / yesAI1 / yesAI2 / notAI2 / println`

The generic AIE is made pending on `f`. When `println` returns, execution continues in `notAI2`, the `RuntimeException` is thrown, and the `finally` block is executed. However, the generic AIE rather than the `RuntimeException` is propagated back to `yesAI2`. Since AIEs are not caught in AI code, the exception is immediately propagated back to the closest dynamically enclosing ATC-deferred section, namely the point after the invocation of `yesAI1` in `notAI1`. The generic AIE is thrown there (still pending), and is handled by the `catch` block in `notAI1`. Since the propagation parameter to `happened` is `false`, the AIE is marked non-pending.[6] Control returns normally to the `run` method, which eventually returns. The following will be the output:

```
Entering try in run
Entering try in notAI1
Entering try in yesAI1
Entering try in yesAI2
Entering try in notAI2
In notAI2, finally
Catching AIE in notAI1
Finishing try in run
```

- `Run / notAI1/ println("Entering try in notAI1")`

[6] This usage of `happened()` assumes that no other attempt to trigger an ATC in `f` will occur while the processing of `f.interrupt()` is in progress. Section 5.4 shows how to deal with nested ATCs.

Here the ATC is triggered before the thread reaches AI code. The generic AIE is marked pending on `f`, and `f` continues normal execution until it invokes `yesAI1`. At this point the AIE is thrown (and it is still pending). Control passes to the `catch` block in `notAI1`, the `happened` method resets the pending status of the AIE, and control returns to `run`. The `run` method returns normally.

Note that if the argument to `happened` were `true` rather than `false`, the effects would have been almost the same. The only difference, which would have been unobservable, is that `run` would have returned with an AIE pending.

5.1.3 Example: Triggering an AIE from an `AsyncEventHandler`

Here is a more realistic example. A realtime thread creates a one-shot timer whose expiry will cause an ATC. This illustrates how an `AsyncEventHandler` can trigger an ATC.

```
class TimeoutExample extends RealtimeThread{
  RealtimeThread target;
  static class AtcTrigger extends AsyncEventHandler{
    AtcTrigger(RealtimeThread target; Runnable logic){
      // target is the thread to be interrupted
      // logic.run is the handler
      AsyncEventHandler( logic );
      this.target=target;
    }
  void interruptibleCode() throws AsynchronouslyInterruptedException{
    ... // interruptible stuff
  }
  public void run(){
    OneShotTimer ost =
      new OneShotTimer(
        new RelativeTime( 5000, 0 ),  // 5 seconds
        new AtcTrigger(  // async event handler
          this,  // target to be interrupted
          new Runnable(){
            public void run(){
              this.target.interrupt();
            }
          )
        )
      );
    try{
      interruptibleCode();
    }
    catch (AsynchronouslyInterruptedException aie){
      ... // code that runs if timer fires
      aie.happened(false);
    }
    finally{
      ost.destroy();
    }
  }
  public static void main(String args[]){
    TimeoutExample te = new TimeoutExample();
```

```
      te.start();
    }
  }
```

If the timer fires before `te` completes its execution, the anonymous `asyncEventHandler` passed to the `OneShotTimer` constructor will be scheduled, and its `run` method executed (by an unknown thread). The effect is to call `interrupt()` on the original thread `te`. If `te` is in `interruptibleCode`, the generic AIE will be thrown, and control will pass to the handler for AIE. This handler suppresses propagation of the AIE. The `finally` clause is then executed; `destroy()` will allow the timer's resources to be reclaimed.

5.1.4 Example: Terminating a RealtimeThread

One of the intended applications of ATC is to terminate a `RealtimeThread` without the dangers of `Thread.stop()` or `Thread.destroy()` but also without the latency induced by polling. The semantics of `interrupt()` allows the programmer to meet this requirement. A typical style is shown in the following example:

```
class AllowTermination extends RealtimeThread{
  void interruptibleCode() throws AsynchronouslyInterruptedException{
    ... // thread algorithm
  }
  public void run(){
    try{
      interruptibleCode();
      ... // here if thread has not been interrupted
    }
    catch( AsynchronouslyInterruptedException aie ){
      ... // here if thread has been interrupted
      aie.happened(false); // stop propagation
    }
  }
}
```

The latency depends on how the `interruptibleCode()` method is written.

5.2 The `Interruptible` Interface

Programming at the level of *`rtthread`*`.interrupt()` and *`aie`*`.happened()` can be error prone. The RTSJ provides a higher-level mechanism that hides the low-level details:

- The interface `Interruptible`, with methods `run()` and `interruptAction()`; each of these methods takes an AIE as parameter
- The methods `doInterruptible()` and `fire()`, in the AIE class; the `doInterruptible` method takes an `Interruptible` as parameter

The anticipated style is for the target thread to construct an instance `aie` of class AIE and to invoke `aie.doInterruptible(i)` where `i` references an object of a class that implements the `Interruptible` interface. Invoking `aie.doInterruptible(i)` causes `i.run(aie)` to be called. Note that this is a *synchronous* call: `aie.doInterruptible(i)` does not return until `i.run(aie)` returns. While control is in `i.run(aie)`, `aie` is enabled. If `aie` is fired before `i.run(aie)` returns, then execution

of the `run()` method is abandoned (as soon as `run()` is in an AI section) and `i.interruptAction(aie)` is invoked.

A source thread needs to trigger this activity by invoking `aie.fire()`. (Thus the source thread needs the reference to the AIE; this would typically be passed in a constructor for the source thread.) Invoking `aie.fire()` too early (before `aie.doInterruptible()` has been called) or too late (after `aie.doInterruptible()` has returned) have no effect. In essence, when a realtime thread `t` invokes `aie.doInterruptible()`, `t`'s current stackframe is marked as the "owner" of the AIE. To avoid implementation complexity, a given AIE can be owned by only one stackframe at a time. If `aie.doInterruptible` is invoked while some other invocation of the same call is in progress, then the second call has no effect[7].

Note that the exception handling and propagation inherent in ATC is transparent to a program that uses `fire()` and `doInterruptible()`. The implementation of `doInterruptible` must manage the asynchronous exception by invoking `interruptAction` to handle the exception and by making the AIE non-pending.

Here is a skeletal example that illustrates this style:

```
class Foo extends RealtimeThread{
  AsynchronouslyInterruptedException aie;

  Foo( AsynchronouslyInterruptedException aie ){
    this.aie = aie;
  }

  public static void run(){
    ... // Noninterruptible code
    aie.doInterruptible(
      new Interruptible(
        public void run(
           AsynchronouslyInterruptedException e )
        throws AsynchronouslyInterruptedException{
          ... // interruptible code
        }
        public void interruptAction(
          AsynchronouslyInterruptedException e ){
          ... // code that executes if run() is interrupted
        }
      )
    );
  }
  public static void main( String[] args ){
    AsynchronouslyInterruptedException aie =
      new AsynchronouslyInterruptedException()
    Foo foo = new Foo( aie );
    Bar bar = new Bar( aie );
    foo.start();
    bar.start();
```

[7] The `doInterruptible` method returns a `boolean` result that normally will be true. It is `false` if another invocation of `doInterruptible` on the same AIE is in progress for any thread.

```
  }
}
class Bar extends RealtimeThread{
  AsynchronouslyInterruptedException aie;

  Bar(AsynchronouslyInterruptedException aie ){
    this.aie = aie;
  }

  public static void run(){
    ...
    aie.fire();  // interrupts foo if foo is in doInterruptible
    ...
  }
}
```

Note that the "throws AIE" clause on the `run()` method defined in the anonymous `Interrupible` class is essential; without it, the method will not be AI.

Using `doInterruptible` allows fine-grained control over asynchronous interruptibility of the `Interruptible`'s `run()` method, in particular the ability to turn any sequence of code into an ATC-deferred section. This is accomplished by calling `e.disable()` and subsequently `e.enable()` from `run()` where `e` is the AIE parameter to `run()`. If `aie.fire()` is invoked while a thread is executing the `run()` method of the `Interruptible` passed to `aie.doInterruptible`, after `disable()` has been invoked but before the matching `enable()` has been reached, then the AIE is made pending. It will be thrown (and stay pending, with the usual semantics) after it is enabled. The `enable()` and `disable()` methods return `boolean` results. Normally the result for each will be `true`; it is `false` if the method is not invoked within a dynamically enclosing call of `doInterruptible` on the same AIE; in such a case invoking `enable()` or `disable()` has no effect.

Here's an example of `disable()` and `enable()`, shown as a variation of the `run()` method from class `Foo` above:

```
public static void run(){
  ... // Noninterruptible code
  aie.doInterruptible(
    new Interruptible(
      public void run(
        AsynchronouslyInterruptedException e)
         throws AsynchronouslyInterruptedException{
        ... // interruptible code
        e.disable();
        ... // non-interruptible code
        e.enable();
        ... // interruptible code
      }
      public void interruptAction(
        AsynchronouslyInterruptedException e ){
        ... // code that executes if run() is interrupted
      }
    )
```

```
      );
    }
```

5.3 The Timed Class

Section 5.1.3 illustrated programming a timeout using `interrupt()` from an `AsyncEventHandler` associated with a `OneShotTimer`. Since timeouts are a common requirement, the RTSJ provides a way to obtain the needed functionality in a notationally more convenient fashion, namely the `Timed` class, which extends AIE. This class has a constructor that takes a `HighResolutionTime` argument. Calling `timed.doInterruptible(i)` on a `Timed` object `timed` supplies in the `Interruptible` parameter `i` both the code that is susceptible to interruption and the code that is to execute in response to the timeout. Here is how the earlier example can be expressed:

```
class TimeoutExample extends RealtimeThread{
  public void run(){
    new Timed(new RelativeTime( 5000, 0 )).
      doInterruptible(
        new Interruptible(){
          public void run(
            AsynchronouslyInterruptedException e)
             throws AsynchronouslyInterruptedException{
            ... // code that is asynchronously interruptible
          }
          public void interruptAction(
            AsynchronouslyInterruptedException e){
            ... // code that runs if timeout occurs
          }
        );
  }
  public static void main(String args[]){
    TimeoutExample te = new TimeoutExample();
    te.start();
  }
}
```

5.4 Nested ATCs

A complication with ATC is to properly deal with the situation where a second ATC is triggered while a first one is in progress. For example, suppose that a realtime thread sets a 500 msec timeout on some AI code. That code calls a method `foo()` that internally sets a 700 msec timeout on another AI section. If the 500 msec expires when the thread is executing in `foo()`, then several things need to happen:

- The 700 msec timer needs to be canceled
- The AIE for the 500 msec timer needs to propagate back to the scope that "owns" it (based on the semantics of 5.1), where it can be handled

These effects are obtained when the programmer uses either `doInterruptible()` or its "syntactic sugar" the `Timed` class. In particular, the RTSJ defines a *level* for each each AIE. When a realtime thread invokes `aie.doInterruptible`, then the level of `aie` is

the dynamic nesting level of the invoking method or constructor. (The `run()` method is at level 1, and any method or constructor directly or indirectly invoked from `run()` has a nesting level 1 greater than that of its invoker.) Before `aie.doInterrupuble` is invoked and after it returns, the nesting level of `aie` is undefined. The "generic AIE" is defined to have level 0.

Suppose that $\texttt{aie}_{new}\texttt{.fire()}$ is invoked while $\texttt{aie}_{old}$ is pending on the thread that invoked $\texttt{aie}_{new}\texttt{.doInterruptible}$. Intuitively, the AIE at the shallower level is the one that should be used. Thus if the level of $\texttt{aie}_{old}$ is less than or equal to the level of $\texttt{aie}_{new}$, $\texttt{aie}_{new}$ is ignored. (This could correspond to a situation where an inner timer goes off while an outer timeout is propagating to its handler.) If the level of $\texttt{aie}_{new}$ is less than the level of $\texttt{aie}_{old}$, then $\texttt{aie}_{new}$ replaces $\texttt{aie}_{old}$ as the pending AIE.

The rules give precedence to the generic AIE triggered by `interrupt()` – it has priority over all other AIEs – since this is typically used to arrange "immediate" thread termination.

Note that there is a potential race condition that programmers need to be alert to. Suppose that a thread is about to handle an AIE, but between the time that the `catch` clause is selected and control enters the associated handler, the AIE has been replaced by another one at a shallower level. The handler should recognize this situation and propagate the new AIE. This effect can be realized through the `aie.happened()` method, which returns a `boolean`: `true` if `aie` is the AIE that is currently pending (the typical situation) but `false` if not (`aie` has been replaced by a different AIE). For example:

```
void foo(){
  try{
    interruptibleCode();  // method that throws AIE
  }
  catch ( AsynchronouslyInterruptedException aie ){
    if (happened(false)){
      // The AIE that was thrown is still the current one
      // Handle it now and turn off its propagation
      ...
    } else {
      // New AIE has occurred, so let it propagate
      AsynchronouslyInterruptedException.propagate();
    }
  }
}
```

The static method `AsynchronouslyInterruptedException.propagate()` keeps the pending status of the currently pending AIE, thus causing it to propagate. Actually this is not needed since by default the new AIE stays pending, but including it makes clearer that the propagation is intentional.

6 Implementation Model

6.1 Implementation Requirements

The implementation of ATC in the Reference Implementation (RI) for the RTSJ is intended only as a proof-of-concept design. Efficiency was a secondary concern relative to its clarity. Indeed, obvious optimizations have been specifically left out of the RI to make the code more clear. Implementors can choose either to use the basic RI design and to add desired improvements, or to use totally different approaches that impose little or no overhead.

6.2 Implementation Design

The basic premise of the RI ATC model is to poll for a flag in the thread context with a low-cost operation after every bytecode. In principle the Java programmer could obtain a similar effect by invoking `Thread.interrupted()` after each statement, but this would be stylistically gruesome and also rather expensive. The RI approach operates at a much finer granularity and with lower cost (a savings of many hundreds of bytecodes compared with explicit user-level polling).

There are always two threads involved in ATC, the *poster* (the thread calling *`aie`*`.fire()` or *`realtimeThread`*`.interrupt()`) and the *target* (the thread that will receive the AIE). Given the polling-based design, the poster does very little work. After acquiring a mutex on the target thread the poster checks to see if an AIE is currently pending on the target. If the poster's AIE is of higher precedence or there is no pending AIE, the reference to the AIE is copied into the target's context as the pending AIE.

The target thread executes a macro after every bytecode that checks for a posted AIE. In this sense, in this implementation, each bytecode instruction can be thought of as comprising a deferred section (because the AIE is initially set to pending) followed by a nondeferred section (the poll, which moves the AIE out of pending). The macro effectively implements:

```
if( <context is nondeferred> ){
  if( <AIE is posted to our context> ){
    <jump to exception handling for this AIE>;
 }
}
```

Note that this approach yields near minimal latency. The target thread will respond to the interrupt after executing at most one intervening bytecode instruction.

The `monitorenter` and `monitorexit` bytecodes, as well as the method invocation bytecodes, have been adjusted to alter the deferred state, associated with the stack frame (the method invocation), as appropriate. Thus the poster need not concern itself with the deferred context of the target.

The method invocation and return bytecodes manage the calling thread's interruptible state by storing the current state in the stack frame at method invocation, and then restoring it on return.

The normal exception handling code has also been altered to reflect the new RTSJ semantics. When this code executes, the AIE is moved from pending to active (so that new AIE's can be posted during AIE propagation). From there, the handling code searches the stack frame for an appropriate `catch` clause, skipping altogether frames that are non-deferred. When an appropriate `catch` block has been found, a reference to the active AIE is pushed onto the stack (which is how exceptions are passed to the catch blocks) and the AIE is moved back to pending, conditional on its precedence over any new AIEs that have been posted to the context. Under race conditions, the object passed to the catch block may be an earlier-propagated AIE versus the AIE currently pending. (See discussion in Section 5.4 above.) The programmer needs to ensure that the `catch` block deals with the appropriate AIE. The AIE remains pending until `happened()` is called appropriately, in which case the reference is deleted, or the code returns to an AI section, when the AIE becomes reactivated and begins propagating. The `propagate()` method in the AIE class is implemented simply as:

```
try{
  throw this;
}
finally{};
```

The `propagate()` method takes advantage of the modified semantics for the `athrow` bytecode, which invokes the special AIE propagation semantics but not the deferral semantics (as the user is synchronously throwing the exception). Control will then transfer to the next outer matching `catch` block.

Conclusions

Asynchronous Transfer of Control is a difficult feature to design, as evidenced by the problems with `Thread.stop()` and `Thread.destroy()` in Java. It is also a difficult feature to use, since at the point where execution resumes the exact state of the program is not known. Nevertheless, for real-time programs ATC is sometimes needed in order to avoid unwanted latency, and thus the RTSJ has chosen to provide explicit support. Very few languages or systems define an ATC mechanism, so there was not a lot of established precedent on which to base the design. The basic RTSJ decision was to capture ATC by specializing the effect of `interrupt()` when applied to a `RealtimeThread`. Instead of simply awakening and throwing an `InterruptedException` at a thread that is blocked, this method throws an `AsynchronouslyInterruptedException` if the thread is executing code that is asynchronously interruptible. For reasons of safety and sound methodology, ATC is deferred in code that is synchronized or that is not in a method or constructor that explicitly indicates (via a throws clause) its readiness to be asynchronously interrupted. The ATC features comprise low-level "building blocks" that can be used for fine-grained control, together with higher-level classes that support common idioms such as timeout.

As the RTSJ is implemented and spreads into practice, it will be interesting to see how the ATC features are used. Although it is not a facility that will be needed in every application, we expect that, when it is required, it will serve its intended purpose for execution predictability / low latency with a style that is readable and a semantics that is consistent with Java design principles.

Bibliography

[Bollella *et. al.* 2000] Greg Bollella, James Gosling, Ben Brosgol, Peter Dibble, Steve Furr, David Hardin, Mark Turnbull; *The Real-Time Specification for Java™*, Addison-Wesley, 2000.

[Chan *et. al.* 1998] Patrick Chan, Rosanna Lee, Douglas Kramer; *The Java™ Class Libraries Second Edition, Volume 1*; Addison-Wesley, 1998.

[Gosling *et. al.* 2000] James Gosling, Bill Joy, Guy Steele, Gilad Bracha; *The Java™ Language Specification Second Edition*, Addison-Wesley, 2000.

[Intermetrics 95] Intermetrics, Inc., *Ada Reference Manual*, International Standard ANSI/ISO/IEC-8652:1995, January 1995.

[JSR-1 2001] Java Specification Request JSR-00001, *The Real-Time Specification for Java™*, November 2001.