Session Summary: Update on the Real-Time Specification for Java

Chair: Ben Brosgol
Rapporteur: Michael González Harbour and Ben Brosgol

1. Introduction

Ben described what's going on with "Real-Time Java" and how it relates to Ada. His presentation comprised three parts:

1) Status update on the Real-Time Java efforts.
2) Technical summary of the Real-Time Specification for Java (RTSJ).
3) Details on the RTSJ's approach to Asynchronous Transfer of Control.

2. Status

Ben summarized the background and current status of the Real-Time Java efforts, focusing mainly on the Sun-sanctioned RTSJ but also briefly mentioning the J-Consortium's "Core Extensions". The RTSJ (V1.0) was officially approved last November (although, interestingly enough, Sun Microsystems' vote was an abstention) with the delivery of the specification (an API), a Reference Implementation, and a Test Compatibility Kit. The latter two items were produced by TimeSys. The spec is currently in maintenance mode, with a V1.1 release planned for Fall 2002; this release is intended to correct errors, resolve ambiguities, and fill in missing details. The Core Extensions document, an alternative approach to real-time Java, proposes a Java-like RTOS with similar functionality to the RTSJ. Jim Moore noted that a J-Consortium Working Group has also produced the JEFF spec, an alternative classfile format that supports storing classes in static memory versus dynamically loading them. There is a possibility that the RTSJ and Core Extensions specs may be merged, but Ben noted that there were licensing issues that would first need to be addressed, and that in the absence of external funding the technical work would probably require around two years to complete.

From one point of view the Real-Time Java efforts compete with Ada, but:

1) Good Ada implementations exist now, whereas the only Real-Time Java implementation at present is the one from TimeSys, and performance was not their goal.
2) From a technical point of view, Ada is much more appropriate than Java for applications that require efficiency or where the traditional static model (compile, link, run) is needed.

Some relevant web sites:

www.rtj.org (Real-Time Specification for Java)
www.j-consortium.org (Core Extensions, plus other J-Consortium information)

3. Technical Summary of the RTSJ

"Vanilla" Java (defined by the language spec) is not appropriate for real-time applications. Some issues:

- Not enough priority levels (10), and no guarantee that priorities are used by the scheduler (i.e. the Java spec allows the implementation to map threads to the thread model provided on the underlying platform).
- Unbounded priority inversions are possible.
- Garbage collector can introduce unpredictable latencies
- All aggregate data in Java go on the garbage-collected heap ("If you sneeze in Java you get a half-dozen heap allocations").
- Asynchrony support is limited.
- Access to the hardware (e.g., dealing with data at particular addresses) is lacking.

The RTSJ has attempted to address these issues by defining an API based on an extension of the Java Thread class. The RTSJ facilities fall into several main areas:

1) The class RealtimeThread and a general scheduling framework.

   The RTSJ supports the concept of a "schedulable object", illustrated by realtime threads and asynchronous event handlers. The implementation needs to provide a traditional preemptive priority-based scheduler (with at least 28 distinct levels, in addition to the 10 for Java threads) but can also supply additional mechanisms such as Earliest Deadline First. Feasibility analysis features are provided, but these are optional (they require per-thread CPU time support, which might not be available on the platform). The spec does not require that a preempted thread go to the head of its ready queue (this is implementation defined), a concession to RTOS vendors that might do things differently.

2) Non Garbage-Collected memory areas.
The RTSJ provides "immortal memory" (where objects are never reclaimed or relocated) and "scoped memory" (a generalization of a stack that is used for objects allocated during the "closure" of a method invocation). These areas are not garbage collected. Assignment rules prevent dangling references. The spec also defines the concept of a "no heap realtime thread" that is not allowed to reference any heap objects and which therefore can preempt the Garbage Collector.

3) Priority inversion control.
Two monitor control policies are provided: priority inheritance (the default) and priority ceiling emulation. The latter does not have the Ada requirement against blocking; thus mutexes / queues are needed in the implementation. The spec allows flexibility, and the user can associate different monitor control policies with different objects (indeed, different monitor control policies with the same object at different points).

4) Time-related functionality.
There are classes related to clocks, timers, etc., including a "rational time" mechanism that allows expressing a periodicity requirement as a frequency (e.g., 6 times per 100 milliseconds).

5) Asynchrony support.
The RTSJ has a mechanism for asynchronous events that is based on the regular Java "event listener" model (the program registers one or more event handlers with an asynchronous event) and that also takes advantage of the general scheduling framework (an asynchronous event handler is a schedulable object, which may or may not have a dedicated thread, based on how the program has defined it). The RTSJ also supplies a feature for Asynchronous Transfer of Control (ATC), described below.

6) Physical Memory and Raw Memory.
There are mechanisms that allow two main functionalities:
   a) The ability to allocate objects in specialized types of memory (e.g. flash), and
   b) The ability to "peek and poke" for primitive data, given an address and an offset.

In summary, the RTSJ opts for predictability, flexibility and generality; the tradeoff is in performance, and some features (e.g. the scope level checks on assignment) will carry run-time costs in the absence of optimizations.

4. Asynchronous Transfer of Control in the RTSJ

Regular Java supplies three methods in the Thread class that relate to ATC:

1) The stop() method, which throws a ThreadDeath exception in a target thread, thus allowing the target thread to do cleanup and to release locks;
2) The destroy() method, which terminates a target thread immediately (no cleanup, no release of locks);
3) The interrupt() method in the Thread class, which awakens a thread that is blocked and throws an InterruptedException.

However:

- The stop() method is inherently dangerous since objects can be left inconsistent, and it might not have the desired effect (terminating the target thread) since the ThreadDeath exception can be caught by a "catch all" handler as the stack unwinds.
- The destroy() method is also dangerous (since locks are not released, other threads may deadlock) and in any event it has never been implemented in any JVM.
- The interrupt() method is basically a polling approach (a thread needs to check whether it has been interrupted).

The RTSJ sought to provide an ATC mechanism without the problems of stop() and destroy(). It did this by generalizing interrupt() to apply to realtime threads even when they are not blocked, and by deferring the effect of an asynchronous interrupt in certain sections. The main features are as follows:

a) A new exception class, AsynchronouslyInterruptedException ("AIE" for short), a subclass of InterruptedException.
b) The concept of an ATC-deferred section: all synchronized code, as well as all methods and constructors that lack a "throws AIE" clause (and the complementary concept of asynchronously interruptible code: anything that is not ATC-deferred).

When t.interrupt() is called on a realtime thread t, the effect depends on where T is currently executing. If in code that is not ATC-deferred, then an AIE is thrown. Otherwise, the AIE is not thrown until t next reaches asynchronously interruptible code.

An Ada-like mechanism (the selective abort, with a conceptual second thread serving as the abortable section) was not adopted by the RTSJ, for several reasons:

- It would have required introducing the notion of dependents of a thread, a concept not otherwise needed.
- The interrupt() mechanism was a natural basis for the design Interestingly, for common cases such as timeout and thread abort, Ada offers a much simpler syntax.

The details of the RTSJ rules for ATC are somewhat complex (e.g. an AIE stays pending even after it has been thrown), and the spec provides some higher-level constructs for common ATC idioms such as timeout.