A Practical Comparison Between Java and Ada in Implementing a Real-Time Embedded System

Eric Potratz
Department of Computer Science
University of Northern Iowa
Cedar Falls, IA 50614-0507
+ 1.319.273.2618
epotratz@forbin.net

ABSTRACT
This paper presents a student's observations from an undergraduate research project that explored using Java to implement the software for a real-time embedded system that was originally implemented in a university-level real-time systems course using Ada 95. It briefly gives an overview of the project, the decision made concerning which Java virtual machine to use, and how that virtual machine performed in the real-time environment. It then goes into detail about the merits and drawbacks of using Java to implement real-time and embedded systems such as this one and how using Java to implement them compares with using Ada.

Categories and Subject Descriptors
D.3.3. [**Programming Languages**]: Language Constructs and Features – *classes and objects; concurrent programming structures; control structures; data types and structures; dynamic storage management; inheritance; procedures, functions, and subroutines*

General Terms
Algorithms, Design, Performance, Languages

Keywords
Ada, concurrency, conditional synchronization, drivers, embedded systems, Java, memory management, object-oriented programming, package elaboration, performance, priority inversion, real-time systems, scheduling

1. INTRODUCTION
The Java Programming Language has come to have a significant role in areas ranging from university-level Computer Science education to implementing computer applications in both desktop and server environments. Interestingly, Java is also growing in popularity as a tool to implement applications in embedded systems environments.

SIGAda'03, December 7–11, 2003, San Diego, California, USA.

After using Ada 95 to develop software for a real-time embedded system in the Real-Time Systems course at the University of Northern Iowa, I chose to take the opportunity in my senior undergraduate research project to explore how usable Java is in implementing that same system. Over the course of implementing this real-time embedded system, a number of notable strengths and weaknesses in the Java language emerged.

2. PROJECT OVERVIEW
The goal for the signature project of UNI's Real-Time Systems course [11] is to write the software for an embedded system that controls an electric model railroad. The basic system specification requires the software to provide users with a way to control multiple trains and prevent collisions between trains operating on the tracks at the same time. Specific real-time and embedded system issues involved in implementing this specification include writing low-level drivers to interface with the hardware, designing and implementing concurrent processes, and satisfying a particular hard real-time requirement that prevents hardware damage at the instant when a train passes from one electrically isolated section of track to another.

For the Real-Time Systems course, we implemented this system using the Ada 95 programming language with the GNAT compiler developed by Ada Core Technologies [16] and used the real-time operating system known as MaRTE OS [9]. The target system which our software had to control includes:

- a 133 MHz Intel x86-compatible AMD K5 microprocessor
- 32 megabytes of RAM
- a VGA video adapter
- a DoubleTalk voice synthesizer (manufactured by RC Systems, Inc.) [6]
- turnouts, which are Y-shaped three-way junctions in the track through which a train can pass between the common lower arm and either the left upper arm or right upper arm at a time depending on which direction it is switched to, controlled by electric motors
- adjustable power supplies called *cabs*, one available for each train operating on the tracks
- Hall effect sensors spaced along the tracks to monitor the movement of trains

- hand controller input devices including pushbuttons, toggle switches and an analog dial that are intended to allow users to control trains

3. CHOOSING A VIRTUAL MACHINE

For the Java-based implementation of this system, I would have ideally liked to use an implementation of the Real-Time Java Expert Group's [12] *Real-Time Specification for Java* [2]. Virtual machines that implement this specification provide additional degrees of flexibility for thread scheduling and memory management helpful in implementing real-time systems that are not found in standard Java. The only available implementation of this specification at the time, though, was the reference implementation provided by TimeSys [15].

TimeSys' Real-Time Java virtual machine only runs with full functionality on TimeSys Linux—a specialized version of Linux developed by TimeSys for real-time applications. Unfortunately, this implementation is not particularly suited for small embedded systems. The size of the TimeSys Linux kernel, the basic utilities required to boot a minimal working Linux system, and the reference implementation's Java virtual machine executable far exceed the amount of storage provided by a 3½-inch disk, which is the medium that we used to transfer the Ada version of the software to the target system. In addition, low-level access to PC hardware in TimeSys Linux involves going through the Linux kernel, introducing extra processing overhead that is not there in MaRTE OS where direct access to the hardware is allowed.

So, instead of this reference implementation, I chose to look for a more compact Java virtual machine—one better suited for a small embedded system—that could run on top of MaRTE OS without a large amount of work. Out of the available portable Java virtual machine implementations designed specifically for real-time or embedded systems, I chose to use SimpleRTJ [5] [13], developed by RTJ Computing Pty. Ltd. Despite its significant lack of real-time functionality compared to TimeSys' Real-Time Java implementation, the features that led me to choose this particular virtual machine include:

- a functional Java environment suitable for an embedded system at little cost (although the version of SimpleRTJ used was just an evaluation version)
- the capability for the operating system, the Java virtual machine, and the Java application to easily fit all on one 3½-inch disk
- the ability to pre-link all classes of a Java application together into a single binary image during the application compilation process, eliminating the need to deal with separate dynamically-loaded class files and a file system to store those files on
- the relatively simple process by which the virtual machine can be ported to the target system using the services provided by MaRTE OS

The SimpleRTJ virtual machine supports Java applications compiled for Sun Microsystems' version 1.1 release of Java. It can execute almost all of the same bytecodes that are used in standard Java 1.1 applets and applications.[1] As a result, all a developer needs in order to compile Java applications for this virtual machine is Sun Microsystems' latest Java SDK [14] and SimpleRTJ's proprietary ClassLinker tool to pre-link the compiled classes.

[1] Only support for the double floating-point primitive type is excluded.

4. JAVA APPLICATION PERFORMANCE

Java programs generally compile to bytecodes, not machine code. A Java virtual machine executing on a target system then interprets these bytecodes to execute Java programs. This method of execution combined with the level of dynamic typing provided by Java's runtime system to support polymorphism, the related lack of support for Java compilers to in-line methods, and Java's memory management model, which relies heavily on dynamically allocated memory and a garbage collector to automatically manage that memory, fosters some curiosity about how these issues affect the performance of Java programs—especially in real-time environments.

While I was unable to port enough of the model railroad software to Java to get trains running over just the course of the semester-long research project, I took the time during the course of my research to try one performance-related test on SimpleRTJ using a critical portion of the software. The test application helped examine how effectively the Java virtual machine could handle interrupts generated by the Hall effect sensors situated along the train tracks.

The train tracks in the model railroad system are divided into a number of electrically isolated sections called *blocks*. Hall effect sensors, which each produce an electric signal at the instant a magnetic field is applied or removed in close proximity, are situated on the boundaries between adjacent blocks. These electric signals are translated into hardware interrupts by the I/O board that the sensors are connected to inside of the computer system. Trains that run on the tracks then have magnets attached to the front of the locomotive and the rear of the caboose. When the trains are in motion, those magnets trigger the Hall sensors, signaling the computer and providing a means by which the computer software can track the location of each train. How quickly the software can respond to these signals is important in determining whether it can satisfy the hard real-time requirement for the system.

The system specification states that only those blocks of track that trains occupy can be kept powered. All other blocks must be unpowered (connected to ground). So, when the front of an electric train locomotive passes from one block into the next adjacent block, power must be applied to the next block before the locomotive's front wheels enter that block (within about 40 to 80 milliseconds of tripping the Hall sensor). This must happen because the contact between the front set of locomotive wheels and the metal rails of the tracks are what supply the locomotive with the electricity it needs to operate. If the computer software cannot switch on power to the next block before the front wheels make contact with them, then the wheels will momentarily close the gap between the two electrically isolated blocks as they pass into the next block, providing a path for electricity to flow between the powered block that the train is currently in and the grounded unpowered block that the train is just entering. This results in a large amount of current flowing through the track hardware, blowing the fuse placed in the circuit to prevent more extensive damage should something like this occur. The hard real-

time requirement for this system requires the computer software to detect a train entering a block and subsequently supply power to that block within that 40 to 80 millisecond time frame. Satisfying this requirement helps prevent these fuses from being blown.

The test program used to help evaluate this aspect of the system was designed to see how well the Java virtual machine could process the interrupts generated by the Hall effect sensors. It would simply speak each Hall sensor's identification number through the DoubleTalk speech synthesizer when it is tripped and its implementation involved both interrupt handlers for the Hall sensors and a driver for the DoubleTalk voice synthesizer whose implementation involved the use of a second thread. The vast majority of the Java version of this program was implemented entirely in Java with only very small portions—those used to access the desired hardware—written in Ada and assembly language.

Upon executing the Java version of the test program and flooding the system with interrupts by rapidly tripping Hall sensors with hand-held magnets, the Java test application appeared to keep up just as well with handling Hall sensor signals as the same test program implemented using Ada. There were no detectable dropped interrupts. In fact, despite the extra overhead that Java has in interpreting bytecodes, the behavior of the Java test program was indistinguishable from the behavior of the Ada test program.

So, a Java virtual machine like SimpleRTJ running on our target system appears to have at least enough raw performance to execute a Java implementation of the model railroad software. The software required to get trains safely running on the tracks, though, involves the use of significantly more concurrent threads of execution. While most of these threads remain idle the majority of the time, some of Java's weaknesses with respect to memory management and thread scheduling (which will be covered in more detail shortly) cast some amount of doubt on whether Java could support the full real-time system without having ported enough of the software to see whether the system could successfully handle a moving train.

5. HELPFUL JAVA LANGUAGE FEATURES

While working on the Java implementation of the model railroad system, a couple of Java language features proved to be very helpful.

5.1 Native Methods

Native methods are one of the language features in Java that are helpful in implementing embedded systems such as the model railroad system. A native method in Java is declared using Java's `native` keyword in the method declaration, and, like a Java abstract method, is not supplied with a method body. Instead, the bodies of native methods are implemented using another language that compiles to the target system's machine code. The machine code for these methods is then either compiled directly into the virtual machine (as is done with SimpleRTJ) or placed in a shared library and dynamically linked to the virtual machine at run-time (as is done by Sun's virtual machine). Whenever a native method is called, this associated machine code is executed instead of Java bytecodes.

Native methods essentially provide the same capability to the Java virtual machine to execute code written in lower-level languages that GNAT provides to Ada in allowing developers to import procedures written in C or C++ and include in-line assembly language in their code. In both Java and Ada, this particular capability is useful when the language does not provide particular low-level access to hardware or operating system services on the target system that another language does. For example, both Java and Ada do not provide access to an Intel x86-compatible microprocessor's I/O port instructions. These, though, are easily accessible using assembly language. Additionally, SimpleRTJ does not provide any console-related services to Java applications. MaRTE OS, though, does provide these services to Ada. Java's native methods provide the means by which Java applications can access hardware features and operating system services such as these that are only accessible through these other languages.

However, I should note that I came across an interesting issue involving how other languages' code executing in a Java application via native methods affects Java virtual machines like SimpleRTJ in a way that does not affect code imported from other languages into an Ada application.

In an Ada application, code written in Ada or imported from another language such as C generally ends up compiled to machine instructions. All concurrent code in the system is then executed in separate threads that are managed by the operating system. In the Java runtime system provided by SimpleRTJ, though, concurrency is implemented differently. SimpleRTJ was designed for use on small embedded systems and, to keep the complexity of the virtual machine relatively low so that it could more effectively run on such systems, the virtual machine was designed so that it executes exclusively in one operating system thread—even when a Java application has multiple Java threads. Instead of using separate threads managed by the operating system to provide support for concurrency, SimpleRTJ relies on an external timer interrupt to tell its single thread when it needs to perform scheduling management for the Java threads in the Java application. Between executing the bytecodes of a Java thread, the virtual machine then looks to see whether that timer interrupt has occurred and performs thread scheduling if it has, switching execution to a different Java thread if necessary. As a result, SimpleRTJ acts much like both an operating system on a single processor system and a single central processing unit whose machine instructions are Java bytecodes all rolled into one thread of execution.

From this difference between the ways that concurrent control is implemented in the Ada runtime system and SimpleRTJ's Java runtime system, the interesting issue with SimpleRTJ and native methods arises out of the differences between how physical CPU machine instructions are designed and what constitutes a Java "instruction" (a bytecode).

Physical CPU machine instructions used in normal execution generally never block. They complete within a finite number of processor cycles. The same behavior extends to most Java bytecodes. Native method "instructions," though, are an exception. In Java virtual machines, a native method call effectively behaves like a single bytecode. If the machine code in

a native method blocks, then the bytecode that invoked the native method effectively blocks that whole bytecode processor in the virtual machine. In the meantime, that bytecode processor cannot execute any other bytecodes because the native method call has blocked it. Since SimpleRTJ is designed so that the bytecode processor and thread scheduler both execute in the same single operating system thread, the part of the virtual machine that manages thread scheduling will also not be able see that timer interrupts are occurring let alone schedule or switch between other Java threads. As a result, even though the Java application itself may have multiple Java threads, none of those threads will execute while one of them has blocked the sole bytecode processor in the virtual machine on a blocking native method.

This problem became apparent when trying to access MaRTE OS's console input services through native methods in a Java application executing in SimpleRTJ. Whenever a line of text was read from the console, the MaRTE OS thread executing the Java virtual machine would block, waiting for either a character or a whole line of input before proceeding. As a result, none of the other Java threads would be scheduled to execute as one would hope, bringing the whole Java application to a grinding halt.

In this particular system implementation, though, SimpleRTJ is fortunately just a piece of software executing on top of an operating system and we can design a way to prevent this from happening. A server task external to the Java virtual machine can be created using a language such as Ada or C that will execute in another thread managed by the operating system. As part of this server task, we can provide Java native methods that allow a Java application to interface with it without blocking the virtual machine. Such native methods can notify the server task of the request and immediately return control to the virtual machine, allowing the virtual machine to continue executing bytecodes while the external task handles the blocking call. In the meantime, the Java thread in the virtual machine that requested the blocking call can use other available native methods to periodically poll that external task while it waits for the request to complete.

Had I the time, I alternatively could have written a console driver in Java using simpler non-blocking native methods that directly accessed the keyboard hardware. Doing so would have solved this problem without resorting to external server tasks. Unfortunately, though, this was beyond the scope of both the model railroad project and my research project.

Additionally, other Java virtual machines may be implemented in a way that does not have this single-thread limitation. Other virtual machine implementations may actually implement concurrent control by producing additional operating-system-level threads for each Java thread in an application. This would effectively provide each Java thread with an independent bytecode processor and separate out the operating system element of the virtual machine that manages thread scheduling so that any one bytecode processor cannot affect it, preventing the blocking problem experienced with SimpleRTJ. Further exploration of this, though, was also beyond the scope of this project.

5.2 Support for Concurrency

Another real-time-related feature of Java very useful in implementing the model railroad software is its direct support for concurrency.

In the Ada implementation of this system, we used Ada language constructs such as *tasks* and *protected types* to respectively implement concurrent control and mutually exclusive regions of execution to protect operations on data shared between tasks. Because Java also directly supports concurrency, porting the model railroad project to Java involved for the most part the use of equivalent language constructs: Thread objects and objects whose operations are protected using synchronized methods.

Thread objects in Java are the equivalents to Ada's tasks. They provide the way to define concurrent execution in Java applications. Each Java Thread running in a Java virtual machine effectively executes concurrently with the other Threads in a Java application. To help protect operations on the data that concurrent Threads manipulate, Java provides a kind of method called a *synchronized method* that is used to implement these operations. These methods are declared by specifying the `synchronized` keyword in the method declaration.

Synchronized methods force a calling Thread to first automatically obtain the lock on the object whose method it wants to execute before it enters that method. Since a particular object's lock can only be held by one particular Thread at a time, synchronized methods help enforce mutual exclusion within an object, providing a way to create Java objects that are basically equivalent to Ada protected types.

This support for concurrency in Java is better integrated into the language than Ada's concurrency support is integrated into Ada, giving Java some advantages. In Ada, tasks and protected types are special types of objects. Declaring and making use of them requires a different syntax than other data types implemented as records and procedures that operate on those records. Tasks and protected types also cannot be extended in an object-oriented manner like objects implemented using Ada's tagged records can. This limits the extent to which Ada's tasks and protected types can be reused.

Being Java objects, though, Java's Threads and protected type equivalents are treated in the same way as all other objects in Java. For example, the syntax for working with these kinds of objects is the same syntax used with all other objects. Threads communicate with other Threads using typical method calls and the syntax for implementing synchronized methods and calling synchronized methods is the same as that for normal methods. Additionally, both Threads and objects that implement equivalents to Ada protected types can be extended in an object oriented manner just like all other objects can. This level of integration can make reusing these kinds of components more straightforward in Java than it is in Ada.

6. JAVA'S DRAWBACKS

Despite Java's convenient support for native methods and concurrency, a host of drawbacks surfaced in the Java language that are significant concerns in implementing real-time embedded systems such as the model railroad system.

6.1 Conditional Synchronization & Task Synchronization

While Ada provides convenient ways to declare entry barriers for conditional synchronization on protected type operations and declare task entry points for task synchronization, Java does not.

It instead provides a more primitive way to implement these kinds of behaviors using a set of low-level methods.

6.1.1 Implementing Entry Barriers

An example from the model railroad project where we decided to use conditional synchronization on a protected type operation is in the DoubleTalk voice synthesizer driver. The implementation of this driver involves a producer-consumer relationship between tasks external to the driver that submit phrases to be spoken through the voice synthesizer and a single task in the driver that sends each of the requested phrases to the DoubleTalk hardware. Since sending a phrase to the hardware is a character by character process that takes a relatively lengthy amount of time and we did not want any producer tasks to have to wait to have their phrase spoken because of the real-time nature of the system, we decided to place all phrases submitted to the driver in a reasonably-sized queue. The task interacting directly with the DoubleTalk hardware then takes phrases out of this queue one by one as they become available and works on sending them through the voice synthesizer.

Because multiple tasks access this queue, we must have mutual exclusion between the parts of the driver that add a phrase to the queue and remove a phrase from the queue. Using Ada, we accomplished this by making the queue type used in the driver a protected type and making the operations that enqueue and dequeue phrases operations on that protected type. In Java, we begin implementing the same kind of behavior by making the methods in the queue class that enqueue and dequeue phrases synchronized methods.

Now by design, we chose to have the DoubleTalk driver start dropping phrases when the queue becomes full. As a result, no conditional synchronization is needed on the procedure that producer threads use to enqueue phrases. When the queue is not full, the call to enqueue a phrase places the phrase in the queue. When the queue is full, the procedure simply discards the phrase since there is nowhere to put it in the queue. Without any need for an entry barrier on this operation, all we have to do in Ada to implement this operation is to make it a procedure operation on the protected queue type. To do the same thing in Java, we need not do anything beyond making the operation a synchronized method in the queue class.

We do want to have conditional synchronization, though, on the operation that the consumer task in the driver uses to remove phrases from the queue. When the phrase queue is empty, we do not want the consumer task to perform a dequeue operation. We would also like to have that task wait in a suspended state while it cannot dequeue a phrase instead of polling the queue to see whether or not it is empty in order to conserve CPU cycles.

To accomplish this using Ada, we have to make the protected queue type's `Dequeue` operation an entry procedure operation and assign it an entry condition stating that the queue must not be empty when a task enters it. Declaring the operation as an entry procedure creates a task queue associated with that operation where tasks waiting to enter that operation will be placed by the runtime system when the entry condition is false. The entry condition specified determines when tasks are to be suspended and placed in the entry queue or allowed to execute the operation.

```
protected type Phrase_Queue_Type is
   entry Dequeue (Phrase : out Phrase_Type);

   procedure Enqueue (Phrase : in Phrase_Type);

   function Is_Empty return Boolean;

   function Is_Full return Boolean;

private
   -- Private declarations...

end Phrase_Queue_Type;


protected body Phrase_Queue_Type is
   entry Dequeue (Phrase : out Phrase_Type)
      when not Is_Empty is
   begin
      -- Dequeue a phrase from the queue,
      -- assigning the dequeued phrase to
      -- 'Phrase'...
   end Dequeue;

   procedure Enqueue (Phrase : in Phrase_Type) is
   begin
      if not Is_Full then
         -- Enqueue 'Phrase'...
      end if;
   end Enqueue;

   function Is_Empty return Boolean is
   begin
      -- Return True when the queue is empty,
      -- False when it is not.
   end Is_Empty;

   function Is_Full return Boolean is
   begin
      -- Return True when the queue is full,
      -- False when it is not.
   end Is_Full;
end Phrase_Queue_Type;
```

Figure 1. The specification and partial implementation of the Ada protected phrase queue type for the Doubletalk driver

Figure 1 shows the partial implementations of the `Enqueue` and `Dequeue` operations for the Ada implementation of this protected queue type. In this implementation, when the consumer task in the DoubleTalk driver calls on the `Dequeue` operation to dequeue a phrase when the queue is empty, the Ada runtime system will suspend that task and place it in the `Dequeue` operation's entry task queue. That task will then stay in that entry queue until some producer task calls the phrase queue's `Enqueue` operation, placing a phrase in the phrase queue and causing the Ada runtime system to reevaluate the `Dequeue` operation's entry condition. When such an action changes the state of the queue so that the queue is no longer empty, the runtime system will wake the consumer task from the `Dequeue` entry queue, allowing it its turn to execute the `Dequeue` operation and remove an existing phrase.

Java does not provide any way to simply declare entry conditions like this on synchronized methods and, as a result, this makes producing this same behavior in Java more complicated. Java

supplies three low-level methods called `wait`, `notify`, and `notifyAll` to all objects and a combination of these methods can be used to implement the entry barrier for this queue class's `dequeue` method.

When a particular Java Thread owns the lock on a particular object and calls that object's `wait` method, it is suspended and placed in the object's set of waiting Threads, one of which is provided for every Java object. It then gives up the lock on that object. At a future point in time, when the object's state has just changed so that an entry condition on the entry barrier has become true, another Thread that owns the object's lock at that time must explicitly call the object's `notify` method. This method "notifies," or wakes up, any one Thread currently in the object's wait set. After such a Thread is awakened, it must then successfully reacquire the object's lock and, once it has done so, it can then start executing again immediately after the call to `wait` that placed it in the wait set.

The partial Java implementation of the entry barrier for this queue class is shown in Figure 2. In this implementation, a Thread entering the `dequeue` method first checks to see whether or not the queue is empty before it attempts to dequeue a phrase. If the queue is empty, it will invoke the queue instance's `wait` method in order to suspend itself, place itself in the queue instance's wait set, and release the queue's object lock. For the purposes of the model railroad project, this call to `wait` is placed in a while loop where the calling Thread stays while the entry condition is false. This loop is placed there just in case the Thread's wait is interrupted for some reason when `wait` throws an `InterruptedException`. This way, if the entry condition happens to still be false when the Thread is awakened due to an interruption, it will keep waiting at the barrier instead of proceeding into the method when it should not.

```
public class PhraseQueue {
   /* Instance variables and constructor
      declarations... */

   public synchronized Phrase dequeue() {
      while (isEmpty()) {
         try {
            wait();
         }
         catch (InterruptedException
                  exception) { }
      }

      /* Dequeue the next phrase in the queue
         and return it */
   }

   public synchronized
      void enqueue(Phrase phrase) {

      if (!isFull()) {
         /* Enqueue 'phrase' ... */
      }

      notify();
   }

   public synchronized boolean isEmpty() {
      /* Return true when the queue is empty,
         false when it is not. */
   }

   public synchronized boolean isFull() {
      /* Return true when the queue is full,
         false when it is not. */
   }
}
```

Figure 2. The partial implementation of the Java phrase queue class for the Doubletalk driver

Finally, something in this implementation must also be in place to wake up this Thread when somebody else puts a phrase in the queue. At the end of the `enqueue` method, a producer Thread must call the queue instance's `notify` method in order to notify the consumer Thread waiting in the queue's wait set that a phrase now exists in the queue, waking that Thread so that it can proceed to dequeue that phrase. If there is no Thread in the wait set, then this call does nothing to the wait set.

Comparing the partial Ada and Java implementations of this queue data type, we see that the Java implementation of entry barriers is more complex than Ada's. In Ada, all we have to do is declare an entry operation and the entry condition for that operation. In Java, though, we have to actually worry about the algorithms involving the `wait` and `notify` methods that will produce the desired behavior. Despite this, this particular Java implementation nevertheless gets the job done in very simple situations like this DoubleTalk phrase queue where only one entry condition exists on one synchronized method.

6.1.2 The Drawbacks of wait, notify, and notifyAll

The queue class used in the implementation of the DoubleTalk driver only uses the `wait` and `notify` methods in a very simple way because there is only one entry barrier on one synchronized method in the class. Within the scope of the model railroad system, this also happens to be the most complicated situation in which we make use of these particular methods. But, it is nevertheless important to note that the `wait`, `notify`, and `notifyAll` methods have a few serious drawbacks when used in more involved ways.

Firstly, compared to Ada, the low-level nature of these methods can make putting additional entry barriers on other Java synchronized methods more than trivial—especially when going from a class where calls to `wait` and `notify` have been optimized for one entry barrier on one synchronized method to multiple entry barriers on multiple synchronized methods.

For example, consider how the queue class implementations in Figures 1 and 2 must be changed if we would decide to add an entry barrier to the queue's enqueue operation so that it would not discard phrases when the queue is full but instead execute the operation only when an opening in the queue is available, suspending calling tasks otherwise. To adjust the Ada implementation shown in Figure 1, all we have to do is change the `Enqueue` procedure of the protected queue type to an entry procedure, creating another task entry queue for that operation, and then add the appropriate entry condition. The Ada runtime system will then automatically take care of ensuring that this additional entry barrier is enforced. The modified Ada implementation including this change is shown in Figure 3.

Adding this entry barrier in Java, though, is not as trivial. Each Java object only has one wait set in which all Threads calling on

```
protected type Phrase_Queue_Type is
   entry Dequeue (Phrase : out Phrase_Type);

   entry Enqueue (Phrase : in Phrase_Type);

   function Is_Empty return Boolean;

   function Is_Full return Boolean;

private
   -- Private declarations...

end Phrase_Queue_Type;


protected body Phrase_Queue_Type is
   entry Dequeue (Phrase : out Phrase_Type)
      when not Is_Empty is
   begin
      -- Dequeue a phrase from the queue,
      -- assigning the dequeued phrase to
      -- 'Phrase'...
   end Dequeue;

   entry Enqueue (Phrase : in Phrase_Type)
      when not Is_Full is
   begin
      -- Enqueue 'Phrase'...
   end Enqueue;

   -- The same implementations for Is_Empty and
   -- Is_Full from Figure 1 are included here
   -- as well...
end Phrase_Queue_Type;
```

Figure 3. The specification and partial implementation of an Ada protected phrase queue type with entry barriers on both the Enqueue and Dequeue operations

`wait` can be placed. So, even if there are multiple synchronized instance methods in a Java class that each have entry barriers with different entry conditions, the Threads waiting on these different entry conditions must still all wait in the same wait set. If this is the case and a particular entry condition on one of the entry barriers becomes true, a call to `notify` then no longer guarantees that a Thread waiting on that particular condition will be the Thread notified. Consequently, the use of the `notify` method in this case can result in latent notification for Threads that actually care about the entry condition that has just changed. If the one Thread awakened by `notify` is waiting on another entry barrier, then the other Threads that actually care about that particular entry condition will have to wait longer—until another future call to `notify`–to be awakened so that they can proceed. Even then there is still no guarantee that one of those particular Threads will be awakened on that next call. This is not desirable behavior–especially in real-time systems.

So, for the entry barrier implementation to behave correctly when an entry condition becomes true, the Java software will now have to either somehow look through all of the Threads in the object's wait set to determine which one waiting on that entry condition should be awakened or awaken all Threads in the object's wait set and leave it up to each one of them to somehow decide whether or not they can proceed. Unfortunately, there is no way in Java to do the former. But, Java provides the `notifyAll` method to help accomplish the latter. When `notifyAll` is called on a particular object, it notifies *every* Thread in that object's wait set. Each Thread must then in turn reacquire the object's lock before it can continue to execute in the synchronized region of code that it was suspended in.

Because every Thread in the wait set is awakened, each must somehow determine whether or not the entry condition at the barrier it is waiting at is true before it proceeds into the method. Those Threads still not allowed to proceed should suspend themselves again by calling `wait`. Those allowed to proceed should continue into the method to execute.

To produce this behavior in the queue class shown in Figure 2, the call to `notify` in the `enqueue` method must be changed to a call to `notifyAll` so that all Threads in the queue instance's wait set can reevaluate their entry conditions. Additionally, a while loop must be inserted at the very beginning of this method with a call to `wait` in its body to enforce the new entry barrier for the `enqueue` operation. This loop will ensure that Threads will only be allowed to enqueue a phrase when the queue is not full.

The `dequeue` method must also be modified to help implement this additional entry barrier. The while loop in place in Figure 2 for the existing `dequeue` entry barrier is already sufficient to prevent Threads from entering the method in situations when they are notified for some change in an entry barrier's entry condition and the queue is still empty. The `dequeue` method, though, must be modified to call the `notifyAll` method after it has dequeued a phrase to notify any Threads waiting in the `enqueue` method that an opening exists in the queue in which they can enqueue a phrase. All of these changes are shown in Figure 4.

Unfortunately, implementing multiple entry barriers in this way in Java generally requires multiple Threads to be awakened when an entry condition becomes true—more than are actually waiting on that particular condition. This introduces extra processing overhead in Java programs due to thread switching that is not present in Ada. When entry conditions are reevaluated in Java, a switch must be made to each waiting Thread so that each can reevaluate its own entry condition and take the appropriate action. In Ada, though, the runtime system takes care of evaluating entry conditions. It does not force a switch to every task waiting at every entry operation on a protected object so that each can reevaluate its entry condition. As a result, the Java implementation of entry barriers is more inefficient—especially when a system must manage large numbers of threads.

So, why not try to implement some way in Java involving the `wait`, `notify`, and `notifyAll` methods that separates Threads waiting on an object out into different wait sets—one for each entry barrier? It turns out that attempting to do this is complicated by another problem inherent in the use of the `wait`, `notify`, and `notifyAll` methods. This problem is nested object lock deadlock, which can occur in a Java system when a Thread calls an object's `wait` method while owning the locks for a number of different object instances.

To attempt to remove the need to notify each and every object waiting on an instance of the queue class in Figure 4 when an entry condition changes, one might try creating two internal objects in each phrase queue instance that are used solely for their

```
public class PhraseQueue
{
   /* Instance variables and constructor
      declarations... */

   public synchronized Phrase dequeue() {
      while (isEmpty()) {
         try {
            wait();
         }
         catch (InterruptedException
                  exception) { }
      }

      Phrase phrase;

      /* Dequeue the next phrase in the queue,
         assigning it to 'phrase'. */

      notifyAll();
      return phrase;
   }

   public synchronized
      void enqueue(Phrase phrase) {

      while (isFull()) {
         try {
            wait();
         }
         catch (InterruptedException
                  exception) { }
      }

      /* Enqueue 'phrase' ... */

      notifyAll();
   }

   /* The same implementations for isEmpty and
      isFull from Figure 2 are included here as
      well...  */
}
```

Figure 4. The partial Java implementation of a phrase queue class with entry barriers on both the enqueue and dequeue operations

wait sets. All Threads waiting on the `enqueue` method's entry barrier could then by some means be placed in one of those objects' wait set and all Threads waiting on the `dequeue` method's entry barrier could be placed in the other object's wait set. Once this is done, then only one Thread waiting on the `enqueue` entry barrier's corresponding wait set object would have to be notified each time a phrase is dequeued since we know for sure that Threads in that object's wait set are waiting on just that particular entry condition. We would not have to notify all Threads waiting at all of the queue instance's entry barriers. The same would hold true for Threads waiting on the `dequeue` entry barrier whenever a phrase is enqueued.

We still, though, must have mutual exclusion between the `enqueue` and `dequeue` methods. To achieve this, both of these methods must still be synchronized so that Threads trying to perform either operation are forced to execute them one at a time. But if this must be the case, consider what will happen when a Thread must be suspended at one of the entry barriers. A Thread entering either the `enqueue` or `dequeue` method must first obtain the phrase queue's object lock. If the entry barrier's entry condition evaluates to false, that Thread must then wait at the barrier until the entry condition becomes true. To suspend itself in the correct wait set, it must next obtain the object lock on the internal object whose wait set is to contain all the Threads waiting at that particular entry barrier. But, when it finally calls the internal wait set object's `wait` method to suspend itself, it gives up only the lock on the internal wait set object. It does *not* release the phrase queue lock it first obtained to enter the synchronized `enqueue` or `dequeue` method. As a result, the Thread will have suspended itself but no other Thread will be able to enqueue or dequeue anything and wake that Thread back up. This will place any Threads that attempt to use that phrase queue in deadlock.

This scenario demonstrates a more general problem in Java that can occur when the `wait`, `notify`, and `notifyAll` methods are used in situations where Threads have made nested calls to obtain the object locks for different object instances. One might hope that a Thread calling `wait` would relinquish all object locks that it possesses and then reacquire each of those locks after it has been notified at a later time, but it only relinquishes the lock on the object that it called the `wait` method on, retaining all other locks it has. Consequently, other Threads will be locked out of all objects whose locks are still possessed by the suspended Thread, vastly increasing the likelihood for deadlock in the system. This is yet another drawback to how the `wait`, `notify`, and `notifyAll` methods behave—especially in systems that use these methods and synchronized methods in more involved ways.

Lastly, even though we designed the model railroad software so that it did not rely on the need to extend concurrency-related objects in Java that use the `wait`, `notify`, and `notifyAll` methods, it is worth noting that the process of extending these kinds of classes can be greatly complicated when a subclass tries to add more functionality that uses these methods. In many cases, while a parent class that uses these methods may work perfectly by itself, subclasses that add functionality using `wait`, `notify`, and `notifyAll` may fail to work as intended due to the way in which the parent and subclass implementations interact with one another. This behavior is generally referred to as the "inheritance anomaly" [3] [4].

In Java, this anomaly can lead to undesired behavior ranging from latent Thread notification to race conditions. Exactly how the class misbehaves depends on the implementations of the parent class and subclass. Most of the time, these problems can only be corrected by refining the implementations of both the parent class and the subclass. This is unfortunate since a developer must be concerned about how the parent class is implemented in order to implement the subclass properly and great care must be taken in designing both a parent class that uses these methods as well as its subclasses. This need for concern about the parent class's implementation also violates encapsulation. So, while one of Java's strengths may be that it allows concurrency-related features like Threads and synchronized methods to be extended in an object-oriented manner, the use of the `wait`, `notify`, and `notifyAll` methods in these kinds of objects has the potential to turn this convenience into more of a nightmare.

6.1.3 Thread Synchronization

The Ada implementation of the model railroad software did not rely much on task synchronization. The majority of the tasks in the software's design were periodic and their behavior relied on

global data structures that they shared access to, so I did not have the opportunity to explore implementing task synchronization in Java in as much depth as entry barriers. The only places where the Ada implementation used task synchronization were where several instances of a particular task type existed concurrently at runtime and each task instance needed to be initialized with information like an identification number before it could execute properly.

A particular example of this in the system is in the implementation of the tasks that control the turnouts on the tracks. These Y-shaped junctions in track allow a train approaching the lower *common* arm of the turnout to choose between going through either the upper *left* or *right* arm. A train approaching either the left or right arm instead must ensure that the turnout is switched to the correct direction so that it can safely travel through to the common arm without derailing. In either case, if the turnout fails to completely switch to the correct direction before the train passes through the junction, the train will derail. To help prevent this, we create a task for each turnout that watches its turnout as it switches directions to make sure that it reaches its desired position within a reasonable amount of time. If, for some reason, the turnout becomes stuck and cannot get to its desired position, then its task will quickly stop any train that is about to go through the turnout and attempt to unstick the turnout by switching the turnout's position from left to right and back over and over again until it successfully reaches its desired position.

Before these tasks can execute properly, each must know which turnout they have to watch. In the Ada implementation of the software, we were prohibited from using dynamic memory and, as a result, could not dynamically create tasks. So, we could not separately create tasks in such a way that we could effectively use type discriminants to assign a turnout to each task. Instead, we had to declare an array of these tasks. Before each task could then proceed to watch its turnout, it had to service an entry point at the very beginning of its block of execution where the main program could synchronize with it and give it its turnout's identification number.

In the Java implementation, since we are forced to use dynamically created objects for Threads, we can go ahead and use the equivalent of an Ada type discriminant by creating a constructor method for the turnout Thread class that requires a turnout's identification number to be supplied when a turnout Thread is created. When each of these Threads is created, its identification number is then simply passed to the constructor and there is no need for any task synchronization.

Despite the lack of utilization of task synchronization in the Java version of this system, it is worth noting the similarities between how task synchronization and entry barriers can be implemented in Java. As with entry barriers, task synchronization involves one task waiting for a particular condition to become true. For task synchronization, that condition is whether or not the other Java Thread involved in the synchronization is at a point in its execution where it is ready to rendezvous. Because of this, the `wait`, `notify`, and `notifyAll` methods can be used to implement blocking task synchronization in a manner similar to implementing entry barriers on synchronized methods. Consequently, implementing task synchronization in this way will have the same types of drawbacks seen in implementing entry barriers.

Particularly, because the Ada runtime system automatically handles the dynamic aspects of task synchronization in addition to handling the enforcement of entry barriers, implementing task synchronization is more complicated in Java than it is in Ada. All a developer has to do in Ada is define entry points for a task and what to do at those points where entries are accepted in the sever task. Then, when the system is executing and either the server or client task wishes to synchronize at that entry point, the runtime system takes care of making sure both tasks are at the proper points in their execution before communication between them is allowed to proceed. In Java, a developer has to program this behavior manually using the `wait`, `notify`, and `notifyAll` methods.

6.2 Thread Scheduling

Standard Java virtual machines provide some scale of control over how Threads can be scheduled while an application is executing. The Java virtual machine provided by Sun Microsystems allows developers to assign priorities to Threads. These priorities are then used by the virtual machine to determine how to switch among Threads while a Java application is running. SimpleRTJ, designed for use in smaller embedded systems, provides much more simplistic support for priorities. In SimpleRTJ, all regular application Threads have the same priority.[2] Only SimpleRTJ's Events Thread, which exists to handle asynchronous events such as hardware interrupts, is given a higher priority over all of the other Threads to guarantee that any time-critical handling of these kinds events is done in a timely manner.

These virtual machines provide a level of support for Thread priorities no matter how limited. But, even though they do this, they still only are implemented to satisfy the Thread scheduling specifications for standard (non-real-time) Java that are laid out in *The Java Language Specification* [8]. As a result, there are two significant concerns with respect to Thread scheduling must be considered while implementing the model railroad software.

6.2.1 Arbitrary Thread Scheduling

One issue with standard Java's Thread scheduling is that *The Java Language Specification* says nothing about how Threads that are waiting to acquire an object's lock are to be scheduled for access to the lock. It also does not address how Threads waiting in an object's wait set are to be scheduled for notification when `notify` is invoked to wake just a single Thread. According to *The Java Language Specification*, any arbitrary Thread can be chosen out of those waiting at a synchronized method to acquire an object lock or those to be taken out of an object's wait set after `notify` is called. The next Thread chosen does not necessarily depend on its priority or how long it has been waiting.

This was not how scheduling was handled in the Ada implementation of the model railroad software. The *Ada 95 Reference Manual* [7] addresses how tasks are to be scheduled in equivalent situations. For example, tasks waiting in entry queues will, by default, be chosen for entry based on order of arrival. When the Ada Real-Time Systems Annex is available, tasks can

[2] Consequently, SimpleRTJ does not fully implement this part of the *The Java Language Specification*.

also be assigned priorities. Those priorities, along with a specifiable task dispatching policy, will then be used to determine how tasks are scheduled.

Standard Java's lack of a specification for scheduling in these situations gives it a significant disadvantage as a useful real-time language. Its arbitrarily specified behavior does not help the process of developing real-time applications when developers need to be able to control or at least know how Threads are being scheduled. The *Real-Time Specification for Java* [2], though, extends Java to provide this kind of functionality. It defines a set of virtual machine improvements and class libraries that can be used to specify Thread scheduling policies for normal Thread switching, for how Threads are chosen to acquire an object's lock to enter a synchronized region of code, and for how Threads waiting in a particular object's wait set are to be chosen for notification by basically turning the wait set into a wait queue.

SimpleRTJ does not provide this way to define scheduling policies. The use of Thread priorities in the model railroad system, though, is only critical for handling the Hall sensor hardware interrupts. These interrupts must be handled as soon after they occur as possible and the Events Thread that SimpleRTJ provides to execute the interrupt handler helps ensure that it is always given the highest priority of all Threads executing on the system. Additionally, this interrupt handler is designed so that it never enters any regions of code where it might be placed in a wait set or compete against other Threads to enter a synchronized method. Lastly, thanks to SimpleRTJ, all the other Threads on the system—which do call synchronized methods and use the `wait`, `notify`, and `notifyAll` methods—all have the same priority. So, the fact that standard Java does not specifically schedule according to priorities in these particular situations is not a significant issue in the implementation of the model railroad system. The arbitrarily specified approach to scheduling, though, is despite the fact that it did not appear to have adverse effects on the performance of the implemented portions of the model railroad software.

6.2.2 Priority Inversion

The second of the concerns related to standard Java's Thread scheduling is priority inversion. Since MaRTE OS [9] provided an implementation of the Ada Real-Time Systems Annex [7], the Ada runtime system automatically took care of limiting the effects of possible priority inversion in the Ada implementation of the model railroad software by using a form of priority inheritance. Each task in Ada is assigned a *base priority*. This is the priority level that a developer can give to a task and the priority at which the task typically executes when it is not in critical regions of code. But, while the system is executing, the task's *active priority* can be raised above this base priority in situations where a higher-priority task is waiting on the task to finish executing a critical region of code. Raising this active priority raises the task's effective priority so that it can finish executing in a critical region as soon as possible, limiting the amount of time that the higher-priority task has to wait.

Standard Java, though, as defined in *The Java Language Specification* [8], does not address priority inversion. As a result, a high priority Thread executing in a virtual machine that satisfies this specification could suddenly find itself blocked, waiting to enter a synchronized method where a lower priority Thread is executing. If other higher-priority Threads were executing on the system, then that high-priority Thread would suddenly find itself blocked for a significant amount of time since the lower-priority process would be preempted by the other higher-priority Threads. This results in priority inversion, where the blocked high priority Thread effectively has the priority of the low priority Thread it is waiting on.

This is yet another feature useful in real-time applications that standard Java fails to implement while Ada does. The *Real-Time Specification for Java* [2], though, also takes steps to address this. It provides a means by which to limit priority inversion among Threads competing to enter synchronized methods by using a form of priority inheritance when a particular Thread scheduling policy provided by the specification is enabled.

Because SimpleRTJ is based more on standard Java, it also lacks the means by which to limit the effects of priority inversion. This, though, is also not a critical issue for the model railroad software because of the way it is designed. We gave all tasks in the system the same priority. The only exception to this was the interrupt handler that processes the Hall sensor interrupts. In the Ada implementation, this interrupt handler got priority over other tasks because it was a native hardware interrupt handler. In SimpleRTJ, this interrupt handler is executed by SimpleRTJ's Events Thread, which has priority over all other Threads executing in the virtual machine. Combining this with the fact that this interrupt handling Thread is never allowed to execute where it could potentially block, the situation will never occur where a higher priority Thread will have to wait for one of the other lower priority Threads to get out of a critical region. Consequently, SimpleRTJ is sufficient for the model railroad project in this respect even though it does not address priority inversion. Had we the need to assign different priorities to Threads, though, this drawback of SimpleRTJ could have been a problem.

6.3 Memory Management

Another one of the weaknesses standard Java has as a real-time language is the way that standard Java virtual machines manage memory. While the runtime environments for Ada and other languages like C and C++ make use of a runtime stack on which any type of data can be stored, the vast majority of interesting data in a Java application can only be allocated dynamically from a heap. "Primitive" types like integers, floating point values, boolean values, and object pointers can go on Java's runtime stack. Any compound data types, though, must be implemented as objects which must reside in memory allocated from Java's heap. This makes writing a useful Java application virtually impossible without using dynamic memory allocation. For life-critical embedded systems prohibited from using dynamic memory for safety reasons, this makes using standard Java altogether impossible.

Moreover, while a Java program can create objects on demand, it cannot explicitly destroy them. Instead, it has to rely on the Java garbage collector to destroy any objects that are no longer in use. This lack of control over memory deallocation is potentially a significant inconvenience in implementing real-time applications. Firstly, the application is not able to destroy unused objects at will. As a result, unused objects may take up memory for some time after they could have been destroyed explicitly at a convenient, predetermined moment. Secondly, while the garbage collector removes these objects, it interferes in the operation of the Threads executing in the virtual machine.

For a hard real-time system, this means that the handling of any external events needing immediate attention might be preempted by the garbage collector. If the garbage collector happens to interrupt normal execution at the wrong moment for too long, then hard real-time deadlines may pass. While one can probably perform some sort of worst-case execution time analysis on both the garbage collector and hard real-time code to ensure that the garbage collector will not cause missed deadlines, the existence of the garbage collector most certainly does not make this analysis any easier.

These concerns are of such importance in the implementation of real-time systems, though, that they are being actively addressed in the Real-Time Java community. Despite the lack of attention given to safety-critical systems in current Java specifications, preliminary efforts are underway at the OpenGroup to set aside a subset of the *Real-Time Specification for Java* that can be tested to such a standard that it can be deemed reliable enough for use in safety-critical systems [1].

Additionally, the J Consortium and the Real-Time Java Working Group are working on a specification for *Real-Time Core Extensions* [10] to Java that help address concerns about memory management. It allows developers of the real-time portions of a Java application to declare objects as *stackable*. The space for these stackable objects is allocated from the run-time stack and, as a result, is not managed by the garbage collector. The Real-Time for Java Expert group is also addressing these same concerns in their *Real-Time Specification for Java* [2]. While objects would still be allocated dynamically, this specification allows special kinds of real-time Threads to exclusively use forms of non-heap memory not managed by the garbage collector. As a result, the garbage collector would not be allowed to preempt these real-time Threads.

6.4 Bit-Shifting Operations

A general issue with the Java language that I encountered while implementing the model railroad software is how it performs bit shifting operations. An important part of developing embedded systems is writing low-level drivers that provide the application with an interface to the hardware of the target system. Often, the implementation of these drivers involves reading or changing single bits in primitive data types such as bytes or words.

In its syntax for record declarations, Ada provides a convenient way to access these single bits. Using Ada, a programmer can define a record and then map its components onto particular bits of a byte or word that is used to internally represent that record within the computer. Once such a bit map is defined, the compiler then handles all the operations needed to read and modify the desired bits.

Java provides no such way to do this. Instead, one has to use Java's bit shifting and bit masking operations to implement the same functionality. This by itself is no huge inconvenience although the implementation is less straightforward. One can still obtain the same functionality provided by Ada by using these bit manipulation operations.

What makes using Java's bit manipulations for this purpose frustrating, though, is a combination of two behaviors. One is that all bit shifting and bit masking operations are done on either 32-bit `int` or 64-bit `long` integer types. This means that an 8-bit `byte` or 16-bit `short` integer value is first cast to a 32-bit `int` integer before the operation is performed. The second behavior is that these 32-bit `int` and 64-bit `long` integer types in Java are signed integer types.

This can cause what seems like unexpected results when using a right unsigned bit shift operation to access single bits in `byte` or `short` integer values. One would hope that a right unsigned bit shift operation performed on a negative byte value would introduce zeros on the left-hand side of the byte despite its negatively-signed value. This, though, is not what happens.

Before a shifting operation is performed on a negative `byte`, that `byte` is first converted to a negative 32-bit integer. Then, when the bit shifting operation is performed, the ones in the upper bits of this negative 32-bit integer are shifted to the right into the lower eight bits. When the result is cast back to a `byte`, we find that the unsigned shift operation effectively produced a signed right shift.

But, however inconvenient and unexpected this may seem, there is a way to work around it. Instead of performing just an unsigned right bit shift on the byte using an expression like the following:

```
(byte)(byteValue >>> 2)
```

one first has to mask out the upper 24 bits of the intermediate 32-bit integer value before shifting to obtain the desired result:

```
(byte)((byteValue & 0xFF) >>> 2)
```

This fix is relatively simple. It still, though, is a bothersome inconvenience that is another potential source for errors—especially when developing low-level drivers that require arithmetic on values obtained using the unsigned right bit shift operation.

6.5 Class "Elaboration"

One final issue with Java concerns what *The Java Language Specification* [8] requires of Java compilers in the application compilation process. In Ada, one can write initialization code for packages that is automatically executed by the runtime environment when the application is started. This initialization code is executed during a process called *package elaboration*. In Java, one can declare `static final` (constant global class-wide) "variables" whose values must be initialized—either with primitive type values or object instances created by calling on other class's object constructors. The Java virtual machine must then initialize the values of these constants before it starts executing a Java application. So in effect, the virtual machine must perform the equivalent of "elaborating" these classes. While an Ada compiler is required to check such code to make sure that circular package dependencies will not produce unintended results when elaboration occurs, the Java compiler does not provide this type of checking.

Problems related to this issue caused considerable amount of consternation while developing the Java version of the electric model railroad software. Fortunately, most of the circular class dependency issues could be resolved by looking at which places in the Ada version of the software had circular dependencies and carefully designing the corresponding Java classes to prevent any problems. Despite these efforts, though, once the number of classes in the project grew beyond a certain point, the Java application all of a sudden would started acting like some of its class constants were not being initialized. Upon further exploration, they in fact really were not being initialized, and,

oddly, this was happening both for classes that did depend on other classes and classes that did not.

So unfortunately it became apparent that it was not safe to assume that the virtual machine could correctly "elaborate" classes at startup. There is a way, though, to work around this problem without relying on a compiler that requires careful thinking. First, one has to move all statements in each class that initialize the class's constants to a class-wide `initialize` method in the class. Then, the main program must explicitly call each class's `initialize` method before using them. Further, the order in which it calls them must be such that it ensures that all classes a particular class depends upon are initialized before that class itself is initialized. Of course, this unfortunately prohibits one from declaring the global "constants" in the affected classes as `final` since the compiler will not allow values to be assigned to any `final` variables other than those assigned to them right where they are declared. Only after taking these steps, though, would everything finally initialize correctly and behave properly.

I should also note that, since this particular initialization problem appeared in classes that did not depend upon any other classes, part of this particular problem could possibly have been with SimpleRTJ and not with Java virtual machines in general. To know for sure, I would have had to perform more testing using other virtual machine implementations to see if the exact same behavior could be duplicated. But nevertheless, this does not discount the fact that extra care must be taken with Java to avoid circular class dependencies because neither the compiler nor the virtual machine may detect them.

7. CONCLUSION

From practical experience with UNI Real-Time Systems model railroad project, Java has shown both merits and drawbacks as a language for implementing real-time embedded systems.

Firstly, despite the fact that Java virtual machines interpret bytecodes instead of directly executing machine code, Java operating environments appear to have the raw performance needed to support real-time applications as long as the virtual machine is efficient enough and the target system is fast enough. But, considering Java's inefficiencies with implementing entry barriers and its arbitrary Thread scheduling in particular situations, a more complete implementation of the model railroad software would have provided a more definitive answer on the adequacy of Java's performance for the complete real-time system.

Additionally, Java provides two language features that are particularly useful in implementing the model real-time embedded railroad system. These features are the ability to provide an interface to target-system-specific resources through native methods and direct integrated language support for concurrency available through Thread objects and synchronized methods.

But, despite these strengths, Java has a number of significant drawbacks. One of these is the low-level nature of the methods that Java provides to implement entry barriers for conditional synchronization on regions of mutual exclusion. These make implementing entry barriers more computationally expensive, complicated, and error-prone than doing the same in Ada. Standard Java also lacks a specification for how to schedule Threads waiting to enter synchronized methods and choosing the next Thread to notify in an object's wait set when the `notify` method is called. Ada's specification does address scheduling behavior in equivalent situations.

Compared to Ada, standard Java also leaves much to be desired in its method of memory management, which relies almost exclusively on dynamic memory allocation and a garbage collector that may interfere with the predictable execution of threads. Java additionally provides a less convenient way to access single bits in primitive data types through bit shifting and bit masking operations, and using the unsigned right bit shift operation to access these bits in primitive types smaller than 32-bit integers can produce what seems like unexpected behavior. This makes the implementation of low-level drivers requiring access to single bits of data more obfuscated. Finally, Java lacks an equivalent to Ada's elaboration checking for the code that initializes classes at application startup. This has the potential to cause unexpected behavior in a Java application if class dependency circularity is not carefully considered.

While these last two issues are mainly just inconveniences and can be worked around with some thought, the issues with arbitrary Thread scheduling and Java's memory management model make Java less than ideal as a language for use in real-time applications. These drawbacks, combined with the work currently being done by the J-Consortium, the OpenGroup, and the Real-Time Java Expert Group show that Java is more of a work in progress as a real-time language. They reinforce the importance, though, of the current efforts being put forth by these groups in the Real-Time Java community to provide real-time extensions to Java that solve these issues and make Java a more viable and reliable alternative for implementing real-time embedded systems.

8. REFERENCES

[1] Bergmann, Joe. *Safety Critical JSR Draft 2 R3*. The Open Group. http://www.opengroup.org/rtforum/uploads/40/2932/SafteyCriticalJSRDraft2_r3.doc (accessed on 20 Sep 2003).

[2] Bollella, Gregory (ed.). *The Real-Time Specification for Java*. Addison-Wesley, Boston, MA, 2000.

[3] Brosgol, Benjamin M. "A Comparison of the Concurrency and Real-Time Features of Ada 95 and Java." *SIGAda '98 Proceedings*. Association for Computing Machinery, Inc., 1998, 175-192.

[4] Burns, Alan, and Andy Wellings. *Real-Time Systems and Programming Languages*. Pearson Education Limited, New York, NY, 2001.

[5] *Developer's Guide to Simple Real Time Java.* RTJ Computing Pty. Ltd. http://rtjcom.com/files/simplertj-1.4.0-doc.zip (accessed on 20 Sep 2003).

[6] *DoubleTalk Voice Synthesizers*. RC Systems, Inc. http://www.rcsys.com/dt.htm (accessed on 10 Aug 2003).

[7] *Ada 95 Reference Manual*. Intermetrics, Inc., Cambridge, MA, 1995. International Standard ISO/IEC 8652:1995.

[8] Joy, Bill (ed.). *The Java Language Specification, Second Edition*. Addison Wesley, Boston, MA, 2000.

[9] *MaRTE OS Home Page.* http://marte.unican.es/ (accessed on 11 Aug 2003).

[10] *Real-Time Core Extensions*. J Consortium, Cupertino, CA, 2000. http://www.j-consortium.org/rtjwg/rtce.1.0.14.pdf (accessed on 20 Sep 2003)

[11] *Real-Time Embedded Systems Lab*. Computer Science Department, University of Northern Iowa. http://www.cs.uni.edu/~mccormic/RealTime/ (accessed on 10 Aug 2003).

[12] *Real-Time Java*. Real-Time for Java Expert Group. http://www.rtj.org/ (accessed on 10 Aug 2003).

[13] *Simple Real Time Java, The.* RTJ Computing Pty. Ltd. http://www.rtjcom.com/ (accessed on 10 Aug 2003).

[14] *Source For Java Technology, The*. Sun Microsystems, Inc. http://java.sun.com/ (accessed on 11 Aug 2003).

[15] *TimeSys Java – Reference Implementation*. TimeSys. http://www.timesys.com/index.cfm?hdr=java_header.cfm&bdy=java_bdy_ri.cfm (accessed on 7 Sep 2003).

[16] *Welcome to Ada Core Technologies*. Ada Core Technologies. http://www.gnat.com/ (accessed on 10 Aug 2003).